Rock E、SuperE 系列可编程控制器

# EMC V2.0 使用手册

（第二版）

北京安控科技发展有限公司
BEIJING ECHO TECHNOLOGIES CO., LTD

## 关于本手册

本使用说明编写适用于北京安控科技发展有限公司生产的 RockE20 系列可编程控制器。EMC 为其编译、连接以及下载调试 C 应用程序。

此说明书详述了 EMC 的使用方法。它由以下几部分组成：

- ✧ 软件的安装
- ✧ 软件环境
- ✧ 控制器的软件特性
- ✧ 建立串行通讯
- ✧ 专用控制 C 函数库
- ✧ 实时操作系统运行机制
- ✧ C 程序的编译、连接及运行

# 目　录

**第一章　概述 ..... 8**

1.1 EMC 简介 ..... 8

1.2 编程环境 ..... 8

1.3 模块化 ..... 9

1.4 汇编语言代码 ..... 9

1.5 程序选项 ..... 9

1.5.1 支持语言特性 ..... 9
1.5.2 串行通讯库 ..... 9
1.5.3 支持时钟/日历 ..... 9
1.5.4 软件定时器 ..... 9
1.5.5 占空比与脉冲输出 ..... 9
1.5.6 看门狗定时器 ..... 9
1.5.7 CRC 校验 ..... 10
1.5.8 标准 I/O 函数 ..... 10

1.6 安装 ..... 10

1.6.1 系统需求 ..... 10
1.6.2 安装方法 ..... 10

**第二章 软件环境 ..... 12**

2.1 软件界面 ..... 12

2.2 标题栏 ..... 12

2.3 菜单栏 ..... 12

2.3.1 文件 ..... 12
2.3.2 编辑 ..... 12
2.3.3 查看 ..... 14
2.3.4 编译 ..... 14
2.3.5 通讯 ..... 14
2.3.6 控制器 ..... 17
2.3.7 帮助 ..... 18

2.4 工具栏 ..... 18

**第三章 控制器的软件特性 ..... 20**

3.1 I/O 数据库和寄存器分配 ..... 20
3.1.1 I/O 数据库寄存器类型 ..... 20
3.1.2 定义寄存器分配 ..... 21
3.2 模拟量 I/O ..... 22
3.2.1 模拟量信号范围 ..... 22
3.3 数字量 I/O ..... 22
3.4 计数器 ..... 22
3.5 定时器 ..... 22
3.5.1 定时器函数 ..... 23
3.6 实时时钟 ..... 24
3.6.1 访问时间与日期 ..... 24
3.6.2 瞬间时钟 ..... 25
3.7 看门狗定时器 ..... 25
3.7.1 看门狗定时器如何运行 ..... 25
3.7.2 使用看门狗定时器 ..... 26
**第四章 建立串行通讯 ..... 27**
4.1 串行 I/O 函数 ..... 27
4.2 默认参数 ..... 27
4.3 调试串行通讯 ..... 28
4.4 使用拨号 MODEM ..... 28
4.4.1 MODEM 初始化 ..... 28
4.4.2 连接远程控制器 ..... 29
**第五章 实时操作系统运行机制 ..... 31**
5.1 任务管理 ..... 31
5.2 资源管理 ..... 31
5.2.1 I/O 系统资源 ..... 31
5.2.2 动态内存资源 ..... 32
5.2.3 MODBUS-PARSER 资源 ..... 32
5.2.4 预防优先权倒置 ..... 32

5.3 内部任务通讯 .....33
5.4 事件通知.....33
5.4.1 系统事件.....33
5.5 错误报告.....34
**第六章专用控制 C 函数库 .....35**
6.1 函数集.....35
6.1.1 校验和函数.....35
6.1.2 系统控制函数.....35
6.1.3 实时操作系统函数.....35
6.1.4 实时时钟函数.....36
6.1.5 串口通信函数.....36
6.1.6 通信协议函数.....37
6.1.7 拨号调制解调器函数.....37
6.1.8 I/O 数据库函数 .....37
6.1.9 I/O 硬件函数 .....38
6.1.10 分配寄存器函数.....38
6.1.11 PID 控制器函数 .....38
6.2 头文件.....38
csum.h .....39
database.h.....39
dialup.h.....39
iohw.h.....41
pid.h .....42
primitiv.h.....43
protocol.h .....45
rassign.h .....48
rtc.h .....48
serial.h.....49
system.h .....52
6.3 函数详细说明 .....53
addRegAssignment .....54
alarmIn .....55
allocate_envelope.....57
auto_pid.....58
check_error .....58
checksum.....58
checkSFTranslationTable.....59

clearAllForcing ..... 60
clear_errors ..... 60
clear_pid ..... 60
clear_protocol_status ..... 61
clearRegAssignment ..... 61
clearSFTranslationTable ..... 62
clearStatusBit ..... 62
clear_tx ..... 63
crc_reverse ..... 63
create_task ..... 63
dbase ..... 66
deallocate_envelope ..... 67
end_application ..... 67
end_task ..... 68
endTimedEvent ..... 68
forceLed ..... 69
getBootType ..... 69
getclock ..... 70
getClockAlarm ..... 70
getForceFlag ..... 71
getOutputsInStopMode ..... 72
getPortCharacteristics ..... 72
get_pid ..... 73
get_port ..... 73
getProgramStatus ..... 74
get_protocol ..... 75
get_protocol_status ..... 76
getSFMapping ..... 77
getSFTranslation ..... 77
get_status ..... 77
getStatusBit ..... 78
getTaskInfo ..... 78
getWakeSource ..... 80
install_handler ..... 81
installClockHandler ..... 83
installExitHandler ..... 85
interruptInput ..... 86
interrupt_signal_event ..... 86
interval ..... 87
jiffy ..... 87
ledGetDefault ..... 88
ledPower ..... 88
ledPowerSwitch ..... 89
ledSetDefault ..... 89

master_message ..... 90
modbusExceptionStatus ..... 93
modbusSlaveID ..... 94
modemDial ..... 94
modemDialEnd ..... 95
modemDialStatus ..... 96
modemInit ..... 97
modemInitEnd ..... 97
modemInitStatus ..... 98
overrideDbase ..... 99
poll_event ..... 100
poll_message ..... 100
poll_resource ..... 101
portConfiguration ..... 102
portIndex ..... 102
portStream ..... 102
pulse ..... 102
pulse_train ..... 104
queue_mode ..... 105
read_timer_info ..... 105
receive_message ..... 106
release_processor ..... 106
release_resource ..... 107
report_error ..... 107
request_resource ..... 108
resetClockAlarm ..... 108
route ..... 109
runLed ..... 109
send_message ..... 110
setBootType ..... 111
setclock ..... 112
setClockAlarm ..... 112
setdbase ..... 113
setDTR ..... 114
setForceFlag ..... 115
setjiffy ..... 116
setOutputsInStopMode ..... 116
set_pid ..... 117
set_port ..... 117
setProgamStatus ..... 118
set_protocl ..... 119
setStatus ..... 120
setStatusBit ..... 120
settimer ..... 121

setWakeSource ..... 121
signal_event ..... 122
sleep ..... 123
start_protocol ..... 124
startup_task ..... 125
startTimedEvent ..... 125
timeout ..... 127
timer ..... 127
wait_event ..... 128
wd_auto ..... 128
wd_manual ..... 129
wd_pulse ..... 129

**第七章 C 程序开发 ..... 131**

7.1 程序结构 ..... 131

7.1.1 主函数结构 ..... 131
7.1.2 启动函数结构 ..... 132
7.1.3 数据存储 ..... 133

7.2 控制器初始化 ..... 133

7.3 下载程序 ..... 134

7.4 执行程序 ..... 134

7.5 C 程序开发举例 ..... 135

7.5.1 编辑 ..... 135
7.5.2 编译与连接 ..... 136
7.5.3 下载与运行 ..... 137

**第八章 程序开发注意事项 ..... 140**

8.1 源程序存放目录 ..... 140

8.1.1 问题原因 ..... 140
8.1.2 纠正此问题 ..... 140

8.2 局部静态变量的使用 ..... 140

8.2.1 问题原因 ..... 140
8.2.2 例子 ..... 140
8.2.3 问题的解决 ..... 141
8.2.4 纠正此问题 ..... 141

# 第一章 概述

北京安控科技有限公司生产的 RockE20 系列 PLC，可用于顺序和过程控制中，广泛用于 SCADA 与 DCS 系统。它可同时运行 C 应用程序与梯形逻辑程序。

公司为了使用户更好地使用 RockE20 系列 PLC 控制器，专门开发了两套软件：EMC 软件用于为控制器编译、连接、以及下载运行 C 应用程序；ELadder 为控制器编写梯形图逻辑程序，并下载到控制器调试运行。ELadder 还可以下载及调试 C 应用程序。

下表列出了 EMC 和 ELadder 的主要功能。ELadder 的详细说明请参考北京安控科技有限公司编写的《ELadder 使用手册》。

| | EMC 主要功能 | ELadder 主要功能 |
|---|---|---|
| 1 | 编译、连接 C 应用程序 | 离线、在线编辑梯形逻辑应用程序 |
| 2 | C 程序下载到控制器 | 梯形图和 C 程序下载到控制器 |
| | | PC 及控制器串口设置 |
| 3 | | 控制器类型设置 |
| 4 | 程序状态显示 | 程序状态显示 |
| 5 | 初始化控制器 | 初始化控制器 |
| 6 | 编辑内存监视器 | 编辑内存监视器 |
| 7 | 在线监控 | 在线监控 |
| 8 | 编辑/强制寄存器及删除强制 | 编辑/强制寄存器 |
| 9 | 帮助 | 控制器锁定及解锁 |
| 10 | | 控制器与 MODEM 的连接调试 |
| 11 | | 从控制器读梯形逻辑程序 |
| 12 | | 帮助 |

## 1.1 EMC 简介

EMC 是一种顺序和过程控制的调试工具，用于编译、连接以及下载 C 应用程序。EMC 在实时控制、数字与模拟量 I/O 方面增加了其它特点。EMC 包括 ANSI C 编译器、C 函数库、实时操作系统和 EMC 程序装载器。

## 1.2 编程环境

C 程序可使用大多数文本编辑器进行编写。但在 RockE20 系列控制器中，只能用 EMC 对程序进行编译、汇编、连接。

## 1.3 模块化

EMC 可以编译多个独立的程序模块。这些模块在连接成最终程序前可单独进行测试。命令文件指定不同的模块如何进行连接。

## 1.4 汇编语言代码

汇编语言源代码可由#asm 与#endasm 直接包含于 C 程序中。用 EMC 的编译器将 C 程序转换为汇编语言，此代码可视并可进行更改。结果代码可与汇编程序连接。

## 1.5 程序选项

C 应用程序可以写入 RAM 或 ROM 中。通常将程序写入 RAM 中。程序可调用操作系统 ROM 中的库程序。RAM 是静态的，因此当编程结束且被安装后，程序仍保留在 RAM 中。应用程序也可写入 EEPROM 中。此时，RAM 用于存储数据。

### 1.5.1 支持语言特性

EMC 使用的编译器与 ANSI C 兼容。EMC 提供代码优化器与汇编器。

除标准 C 运算、数据类型与库函数外，EMC 还提供了一组专门用于控制应用的程序。

### 1.5.2 串行通讯库

串行通讯库提供了 ASCII 通讯、通讯协议与串口配置。默认的通讯模式使用 Modbus RTU 通讯协议。它支持访问 I/O 数据库、串口的重新配置与程序上下载。

应用程序可不使能 Modbus 协议，并可将串口作其它用途使用。

### 1.5.3 支持时钟/日历

EMC 支持处理器的硬件时钟日历。时间、日期与星期可由应用软件进行读取与设置。

### 1.5.4 软件定时器

控制器提供 32 个独立的软件定时器。其滴答速率可编程设定，范围从每秒 10 次到每 25.5 秒 1 次 。定时器与数字量输出连接可使外设在设定时间之后开启或关断。所有定时器运行在主系统时钟产生的硬件中断的后台中。

### 1.5.5 占空比与脉冲输出

数字量 I/O 驱动器提供占空比与脉冲输出。占空比输出产生连续方波。脉冲输出产生有限个脉冲序列。输出的产生不依赖于应用程序。

### 1.5.6 看门狗定时器

控制器支持硬件看门狗定时器，用于侦测和响应硬件与软件错误。看门狗定时器触发脉冲由用户程序或系统时钟产生。

## 1.5.7 CRC 校验

为简化自测通讯运算的执行，EMC 提供四种检验和：加法、CRC-16、CRC-CCITT 与按字节进行异或。CRC 运算非常可靠，提供各种多项式方法来侦测几乎所有的通讯错误。其它类型的校验和通过使用库函数非常容易执行。

## 1.5.8 标准 I/O 函数

EMC 提供了多数常用的 C 编程方法。

C 工具函数库支持所有的标准 I/O 函数。控制器没有磁盘驱动器或外设。因此，多数文件处理函数返回固定的响应，表明运行不能被执行。所有设备可由运行系统自动打开，不能关闭。例行程序函数可用于重定向 stdin、stdout、stderr。

# 1.6 安装

## 1.6.1 系统需求

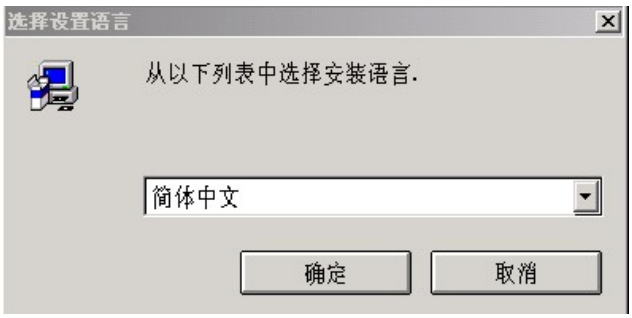
EMC 需要的最小系统配置为：

- ✧ 80386 或 80386 以上的 PC 机。
- ✧ WIN3.1 或 WIN3.1 以上的版本。
- ✧ 最小内存为 4M。
- ✧ 鼠标或其它点击外设。
- ✧ 最小硬盘自由空间为 25M。

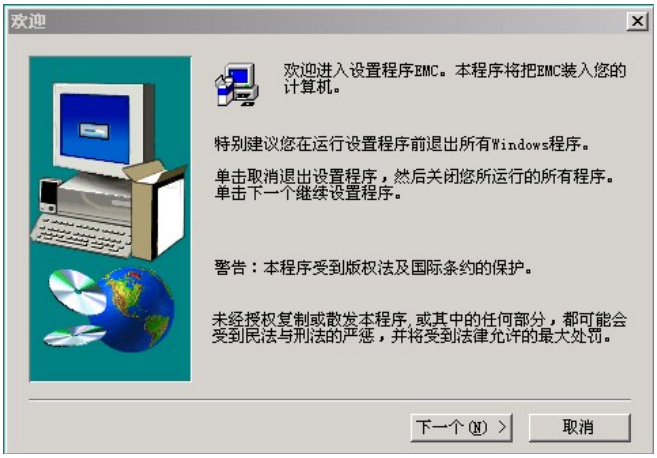
## 1.6.2 安装方法

插入安装光盘，选择安装目录下\EMC-SETUP\DISK\SETUP.EXE 文件，运行安装。

1、 选择语言



2、选择下一个




3、选择程序文件夹，默认即可。

4、安装结束，在开始菜单中有 EMC 选项。

# 第二章 软件环境

EMC 拥有简洁、明晰的软件环境，使用户能方便编译、连接以及下载控制器 C 程序（***不提供编辑功能***）。

## 2.1 软件界面

EMC 程序界面如下图所示。

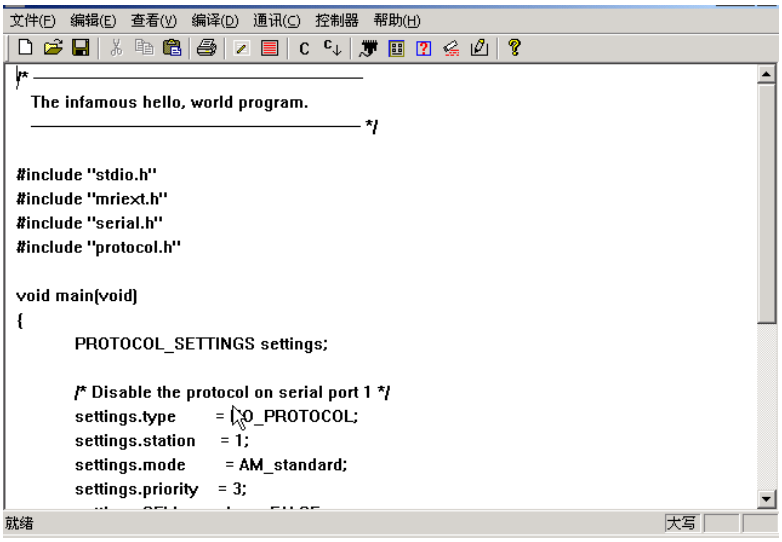
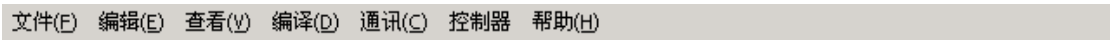



图 2-1

## 2.2 标题栏

标题条显示 EMC 程序标题以及打开 C 程序文件的名称。如果无文件打开，或者有保存新文件，文件名区域为空白。

## 2.3 菜单栏




菜单栏显示了编辑器的程序编辑，调试以及运用等功能。

## 2.3.1 文件

文件菜单含有可以生成、打开和保存 C 程序文件的命令；以及配置打印机选项以及打印 C 程序的命令。各命令操作同 WINDOWS 常规文件操作。

***注：用户只能把编好 C 程序文件用文件或者文件夹的形式保存在 c:\SEMC 下面后才能进行编译，放在其它的目录下的 C 程序文件不能进行编译。***

## 2.3.2 编辑

编辑菜单命令用于查找、复制（***注：EMC 不提供编辑功能***。）

## ✧ 全选

选择编辑区内全部内容。

## ✧ 查找

图 2-3

## ✧ 编辑内存监视器

起始地址选项将确定需要监控的一组寄存器当中的第一个寄存器。终止地址选项将确定一组寄存器当中的最后一个寄存器。把该值设定为 0 即可确定单个寄存器。一组寄存器必须是相同类型（亦即：均以相同的数字 0xxxx、1xxxx、3xxxx 或 4xxxx 开始）。如果已键入该组寄存器，则点击增加>> 按钮即可将所选定的寄存器添加至监控器表并在监控器框内显示它们。

欲从监控器表中删除寄存器，选择该寄存器或监控器框中的寄存器，并点击删除按钮即可。这样，所选定的寄存器将从监控器表以及监控器框中删除。欲清除监控器表并清空监控器框，点击全部删除按钮即可。

图 2-5

显示方式选项将确定寄存器的显示方式。如果该值始终是正数，则使用无符号。如果该值有可能是负数，则使用有符号。

有符号(S)显示寄存器内的值是有符号的数字。如果寄存器内的值有可能为负数，则使用此格式。

无符号(U)显示寄存器内的值是无符号的数字。如果寄存器内的值始终为正数，则使用此格式。

欲改变监控器表内的某寄存器的显示方式，先选择需要改变的寄存器，再选择格式，最后，点击设置按钮以便在监控器表内进行变更。

如果为监控器表添加寄存器，则它们将与当前所选定的显示格式一同添加。

## 2.3.3 查看

### ✧ 工具栏

显示工具栏命令。

### ✧ 状态栏

显示状态栏命令。

## 2.3.4 编译

### ✧ 编译

编译命令将源代码转换为控制器可执行代码，生成 ABS 文件。

***注：编译器与连接器运行于 Microsoft MS-DOS 环境下，编译后按任意键关闭即可。***

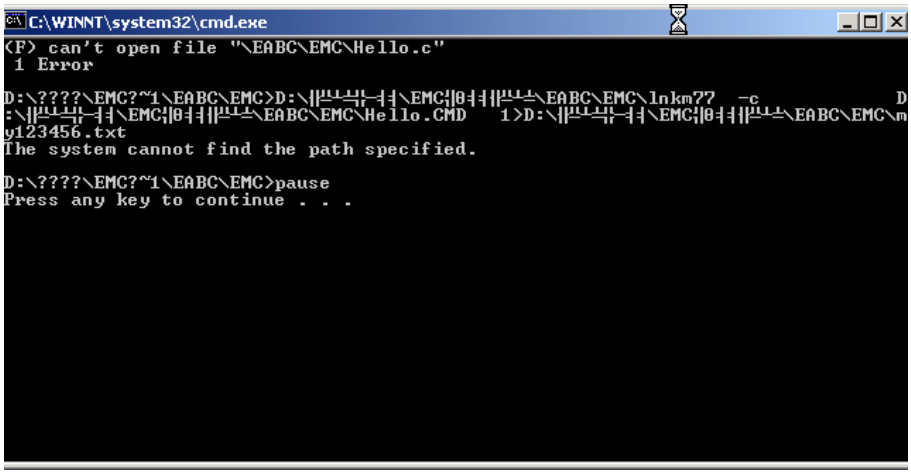



图 2-6

### ✧ RAM

编译连接后生成的可执行程序下载到 RAM。

### ✧ Flash

编译连接后生成的可执行程序下载到 Flash。

## 2.3.5 通讯

用于 PC 机下载 C 程序到控制器。

## ✧ C 程序下载

下载 C 程序到控制器。

图 2-7

这里要注意：下载位置（RAM 或 Flash）应与编译菜单中选择的位置一致。

点击下载命令按钮后，能正常下载出现的对话框如下图所示。

图 2-8

下载完毕后，

图 2-9

## ➢ PC 串口设置

图 2-10

设置 PC 使用的通信参数。

串口将确定 PC 通信串口。

协议将确定通信协议。目前只支持 Modbus RTU 协议和 Modbus ASCII。

波特率将确定波特率。所支持的波特率分别为：300、600、1200、2400、4800、 9600、19200、38400 以及 57600。

奇偶校验将确定校验极性。

停止位数将确定停止位数：1 或 2。

站编辑框将设定控制器的站号数。如果使用标准编址，则有效范围为 1 至 255；如果使用扩展编址，则有效范围为 1 至 65534。

超时将确定 EMC 在重试或显示错误信息之前等待控制器应答所需的时间长度，按秒计算。输入从 1 至 255 的值。缺省值为 3 秒。

## ✧ 程序状态

点击刷新即可看到当前存储器内的程序状态。

图 2-12

## 2.3.6 控制器

### ➢ 初始化控制器

初始化控制器命令可用来把控制器恢复到缺省设置。通常在用控制器启动项目时才这么做。初始化对话框将在选择该命令时出现。

图 2-13

选中初始化控制器选项以便把 I/O 数据库内的所有值复位至缺省设置，包括串口配置寄存器。此命令也会删除所有的强制设定。如果您正在使用缺省以外的设置进行通信，则初始化控制器命令下达之际，PC 串口设置也会改变为缺省设置。

选中擦除寄存器分配表选项以便删除控制器内的所有寄存器分配。

选中擦除 RAM 中梯形逻辑选项以便删除控制器内 RAM 中的逻辑梯形图程序。

选中擦除 Flash 中梯形逻辑选项以便删除控制器内 Flash 中的逻辑梯形图程序。

选中擦除 RAM 中 C 程序以便删除控制器内 RAM 中的 C 应用程序。

选中擦除 Flash 中 C 程序以便删除控制器内 Flash 中的 C 应用程序。

点击清除按钮以便执行所需的初始化。

点击确定按钮以便退出，且不执行任何动作。

### ➢ 内存监视器窗口

编辑菜单中的编辑内存监视器命令所定义的寄存器显示在此窗口。

点击开始按钮，实时显示各地址的值的变化情况。

刷新周期可以更改。

点击关闭按钮关闭对话框退出

图 2-14

- **编辑/强制寄存器**

编辑/强制寄存器命令将显示强制设定寄存器对话框。用户可以查看并编辑寄存器。

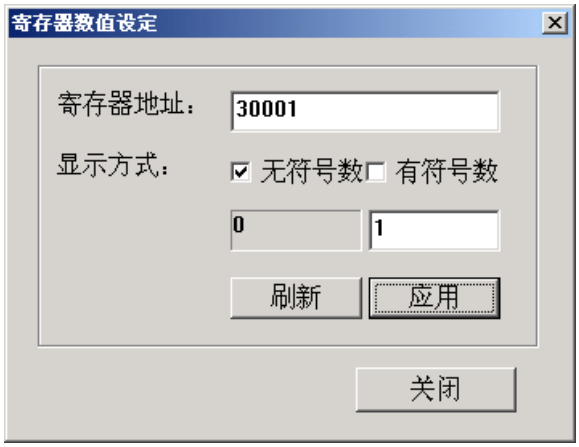



图 2-15

在寄存器地址项中填入任何想查看的寄存器地址，按刷新即可显示其当前值（如灰文本框所示）。可在白文本框内键入需设定的值，点击应用即可。

选择关闭按钮即可关闭对话框。

## 2.3.7 帮助

帮助菜单命令显示有关各种议题的帮助信息。从如下命令中进行选择即可获得更多信息：

- **关于 EMC**

## 2.4 工具栏

工具栏包括了常用的几个命令按钮。

**打开文件**

用于打开已经存在的梯形图程序文件，同文件菜单的打开命令。

**复制**

复制选中内容到剪贴板。

打印

用于打印当前打开的梯形图程序文件，同文件菜单的打印命令。编辑器处于在线调试模式时，此命令变灰。

编辑内存监视器

同编辑菜单的编辑内存监视器命令。

内存监视器窗口

同控制器菜单的内存监视器窗口命令。

编译

同编译菜单的编译命令。

下载

同通讯菜单的 C 程序下载命令。

**PC 端口设置**

同通讯菜单的 PC 端口设置命令。

显示程序状态

同通讯菜单的程序状态命令。

**擦除 C 程序**

同控制器菜单的初始化控制器命令。

**编辑/强制寄存器**

同控制器菜单的编辑/强制寄存器命令。

关于

同帮助菜单的关于 EMC 命令。

# 第三章 控制器的软件特性

RockE20 系列控制器支持专线、电话线和电台无线通讯。可用梯形逻辑和 C 语言编程。本章描述了该控制器的软件特性。

## 3.1 I/O 数据库和寄存器分配

I/O 数据库允许数据在 C 程序、梯形逻辑程序和通讯协议间共享。下图所示为 I/O 数据库的简单图表。

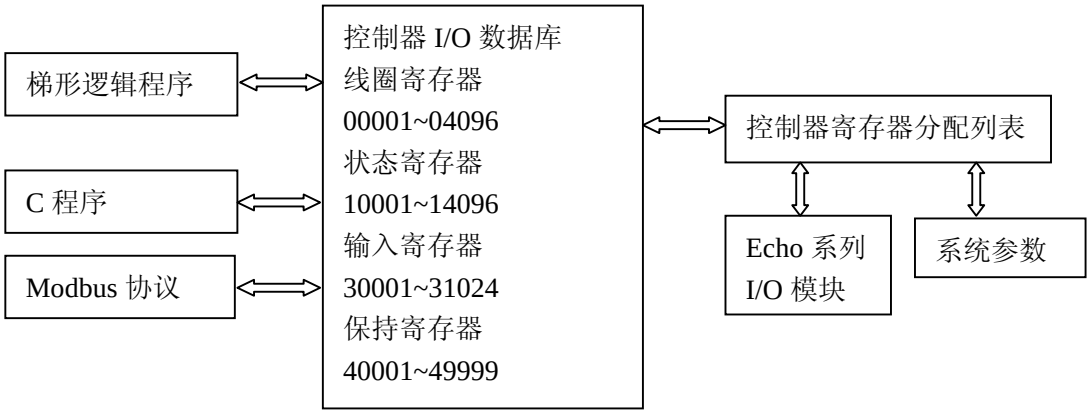



图 3-1

I/O 数据库包括通用寄存器与用户分配寄存器。通用寄存器用于梯形逻辑和 C 应用程序存储过程信息并从远程设备接收信息。最初，I/O 数据库中所有寄存器均为通用寄存器。

用户定义的寄存器直接从 I/O 数据库映像物理的 I/O 硬件或控制器系统配置与诊断参数。由寄存器分配完成从 I/O 数据库到物理的 I/O 硬件与系统参数的寄存器映像。

当控制器复位时用户分配寄存器被初始化为默认的硬件状态或系统参数。当掉电时,分配的输出寄存器不保留其值。当程序启动时，分配的输出寄存器保持其值。

通用目的寄存器在掉电保留其值。只有被应用程序或通讯协议写入时，值才发生改变。

Modbus 通讯协议为控制器提供了标准的通讯接口。Modbus 协议包括 Modbus RTU 和 Modbus ASCII 协议兼容，并可完全访问控制器的 I/O 数据库。

### 3.1.1 I/O 数据库寄存器类型

I/O 数据库将寄存器分为 4 种类型。每种被控制器初始化为通用目的寄存器。

#### ➢ 线圈寄存器

线圈寄存器为位寄存器，位于 I/O 数据库的数字量输出部分。线圈寄存器可通过寄存器分配分配给 Echo 数字量输出模块或 RockE20 系列 I/O 模块。线圈寄存器也可分配给控制器主板数字量输出与系统配置模块。

共有 4096 个线圈寄存器，编号为 00001~04096。梯形逻辑程序、C 程序和 MODBUS

协议可读写这些寄存器。

### ➢ 状态寄存器

状态寄存器是位寄存器，位于 I/O 数据库的数字量输入部分。状态寄存器可通过寄存器分配分配给 Echo 系列数字量输出模块或 RockE20 系列 I/O 模块。状态寄存器也可分配给控制器主板数字量输出与系统诊断模块。

共有 4096 个状态寄存器，编号为 10001~14096。梯形逻辑程序、C 程序和 MODBUS 协议只可读这些寄存器。

### ➢ 输入寄存器

输入寄存器为 16 位寄存器。位于 I/O 数据库的模拟量输入部分。输入寄存器可通过寄存器分配分配给 Echo 系列数字量输出模块或 RockE20 系列 I/O 模块。输入寄存器也可分配给控制器内部模拟量输入与系统诊断模块。

共有 1024 个输入寄存器，编号为 30001~31024。梯形逻辑程序和 MODBUS 协议只可读这些寄存器。C 程序可对这些寄存器进行读写。

### ➢ 保持寄存器

保持寄存器为 16 位寄存器。位于 I/O 数据库的模拟量输出部分。保持寄存器可通过寄存器分配分配给 Echo 数字量输出模块或 RockE20 系列 I/O 模块。保持寄存器也可分配给系统诊断与配置模块。

共有 9999 个保持寄存器，编号为 40001~49999。可以通过梯形逻辑程序、C 语言程序与 Modbus 协议对这些寄存器进行读写。

## 3.1.2 定义寄存器分配

控制器所有 I/O 硬件模块必须分配到 I/O 数据库寄存器中，才能被连续扫描。C 程序通过 I/O 数据库中的用户分配寄存器对 I/O 硬件进行数据读写。

寄存器分配能将 I/O 数据库寄存器分配给使用 I/O 模块。I/O 模块可以是一个真实的 I/O 硬件模块，也可以是一系列控制器参数，如串口设置参数。

用 ELadder 寄存器分配对话框配置的寄存器分配可存储在 ELadder 程序文件中，或直接下载到控制器中。寄存器分配也可使用 C 函数 clearRegAssignment 与 addRegAssignment 在 C 程序中完成。

为得到误差校验防止无效的寄存器分配，使用 ELadder 寄存器分配对话框建立寄存器分配。将寄存器分配保存为梯形逻辑文件，用 C 程序下载或使用 clearRegAssignment 与 addRegAssignment 函数将寄存器定义移到 C 程序中。

## 3.2 模拟量 I/O

### 3.2.1 模拟量信号范围

控制器上的模拟量输入使用 16 位有符号整数格式。此数值范围与实际电压和电流成比例。例如：在+/-10V 模拟量输入范围内，电压和模拟量输入函数返回值的关系为：

| 电压 | Ain 值 |
|---|---|
| -10V | -32767 |
| -5V | -16383 |
| 0V | 0 |
| +5V | 16383 |
| +10V | 32767 |

可添加模拟量 I/O 模块到寄存器分配中，用 dbase 和 setdbase 函数通过 I/O 数据库从模块读写。

## 3.3 数字量 I/O

数字量控制常用于继电器逻辑系统、固态系统或可编程控制器中。数字量控制也可在控制器中执行。控制器可作为可编程控制器使用，提供更多的灵活性。

可添加数字量 I/O 模块到寄存器分配中，用 dbase 和 setdbase 函数通过 I/O 数据库从模块读写。

## 3.4 计数器

控制器支持内部和外部计数器模块。计数器寄存器为 32 位，最大计数值为 4,294,967,295。在下一次计数时返回到 0。

可添加计数器输入模块到寄存器分配中，或用 dbase 和 setdbase 函数通过 I/O 数据库从模块读取。

## 3.5 定时器

控制器提供 32 个功能强大的软件定时器。可用于：

- 产生延时
- 过程时间的计时
- 产生时间中断
- 用时间周期控制数字量输出

32 个定时器独立进行编程，从每秒 10 次到每 25.5 秒一次。时间周期从 0.1 秒到大于 19 天均可被测量与控制。

所有的定时器运行在由主系统时钟产生的硬件中断的后台中。一旦被下载，定时器便开始计数，而不受主程序的影响。

## 3.5.1 定时器函数

有 6 种关于定时器的库函数。可参考 C 函数库章节得到详尽的描述。

intreval　设置滴答时间间隔为十分之一秒

settimer　设置定时器。定时器从设定值向下计数到零。

timmer　读保留在定时器中的时间周期。

**程序举例：**

**例 1：**

```
interval(0,10);
request_resource(IO_SYSTEM):
setdbase(MODBUS,1,1);
release_resource(IO_SYSTEM);
settimer(0,5);
while(timer(0))
{
    release_processor();
}
request_resource(IO_SYSTEM);
setdbase(MODBUS,1,0);
release_resource(IO_SYSTEM);
```

**例 2：**

```
interval(0,1);
request_resource(IO_SYSTEM);
if (dbase(MODBUS,10001))
{
    settimer(0,63000);
    while(dbase(MODBUS,10001))
    {
        release_resource(IO_SYSTEM);
        release_processor();
        request_resource(IO_SYSTEM);
    }
    printf("time period = %u\r\n",63000-timer(0));
}
release_resorce(IO_SYSTEM);
```

**例3：**

```
interval(0,10);
request_resource(IO_SYSTEM);
setdbse(MODBUS,1,1);
settimer(0,60);
```

```
while((dbase(MODBUS,10001)==0) && timer(0))
{
    release_resource(IO_SYSTEM);
    release_processor();
    request_resource(IO_SYSTEM);
}
if (dbase(MODBUS,10001)==0)
    puts("tank is not filling!!\r\n");
else
    puts("tank full\r\n");

setdbase(MODBUS,1,0);
release_resource(IO_SYSTEM);
```

## 3.6 实时时钟

控制器提供了一个实时时钟用来保留运行系统的时间与日期。在掉电过程中，时间与日期仍保持精确。这使得控制器与真实时间保持同步。在周末或假期中，日历可自动使控制器离线，并可自动处理闰年。

### 3.6.1 访问时间与日期

有两个库函数用于访问实时时钟。

**setclock** 设置实时时钟

**getclock** 读取实时时钟

这两个函数引用了定义为头文件 rtc.h 的 clock 结构。

```
struct clock{
    char year;
    char month;
    char day;
    char dayofweek;
    char hour;
    char minute;
    char second;
}
```

下面程序例举了如何设置并显示日期与时间。时钟结构的所有区段需用有效值进行设置。

```
#include “rtc.h”
#include “primitiv.h”
#include “stdio.h”

void main(void)
{
    struct clock now;
```

```
    now.hour = 12;
    now.minute = 1;
    now.second = 0;
    now.day = 1;
    now.month = 1;
    ow.year = 94;
    ow.dayofweek = 6;

    request_resource(IO_SYSTEM);
    setclock(&now);

    now = getclock();
    release_resource(IO_SYSTEM);

    printf("%2d:%2d:%2d",now.hour,now.minute,now.second);
}
```

### 3.6.2 瞬间时钟

瞬间时钟是一个计数器，每秒增加 60 次。瞬时时钟用于测量运行周期或产生延时。为控制器每供电一次时钟便进行一次复位。当计数值达到 5183999 时，时钟便清零。

有两个库函数用于访问实时时钟。

setjiffy　设置瞬时时钟

jiffy　　读取瞬时时钟

## 3.7 看门狗定时器

看门狗定时器可以高速侦测计算机硬件与软件的问题。当出现问题时，CPU 复位，应用程序重新启动。

控制器提供了一个内部看门狗定时器来确保可靠运行。若硬件或系统固件侦测到问题时，看门狗定时器便将 CPU 复位。用户程序可以控制看门狗定时器，侦测出程序的非正常运行。

### 3.7.1 看门狗定时器如何运行

看门狗定时器是一个可再触发，时间延时的定时器。每收到一个复位脉冲便重新开始一个计时序列。若复位脉冲停止，看门狗定时器溢出，表示出现故障。控制器的定时器输出 CPU 复位信号，并关闭 I/O 系统中所有输出。

看门狗定时器通常由操作系统复位。这对应用程序是透明的。运行在此方式下，看门狗定时器可侦测任何硬件或固件问题。

## 3.7.2 使用看门狗定时器

看门狗定时器可侦测应用程序的问题。程序控制定时器并对其进行定期复位。若程序运行不正常，则复位脉冲停止 ，看门狗定时器复位 CPU，程序重新启动。

有三个库函数用于看门狗定时器：

wd-auto　　用看门狗定时器控制操作系统

wd-manual　用看门狗定时器控制应用程序

wd-pulse　　产生看门狗复位脉冲

每 500ms 产生一个看门狗复位脉冲。若看门狗定时器未被复位，则 CPU 复位并且程序重新运行。下面的程序段例举了如何用看门狗定时器侦测程序错误。

```
wd_manual():
do {
    wd_pulse();
}
while (condition)
wd_auto();
```

# 第四章 建立串行通讯

RockE20 系列主控制器 S16-M1 自带两个 RS-232 串行通讯口。串口需配置波特率、数据位、停止位、奇偶校验与通讯协议。

## 4.1 串行 I/O 函数

下面的函数用于串行通讯。

fgetc从流中读取一个字符

fgets从流中读取一个字符串

fputc 写一个字符到流中

fputs 写一个字符串到流中

fread 从流中读取

fwrite 写到流中

getc 从流中读取一个字符

getchar 从标准输入设备中读取一个字符

gets 从流中读取一个字符串

get-port 得到串口配置

initport 重新初始化串口

printf 格式化输出到流

putc 写一个字符到流

putchar 读字符到标准化输出设备

puts 写一个字符串到流

scanf 从流得到一个格式化的输入

set-port 设置串口

## 4.2 默认参数

当控制器在 SERVICE 模式下加电，所有的端口用默认的参数进行复位配置。当控制器在 RUN 模式下加电时，串口使用保存的参数。下表为默认的参数：

| 参数 | COM1 | COM2 |
|---|---|---|
| 波特率 | 9600 | 9600 |
| 奇偶校验 | 无 | 无 |
| 数据位 | 8 | 8 |
| 停止位 | 1 | 1 |
| 通讯方式 | 全双工 | 全双工 |
| 协议 | MODBUS RTU | MODBUS RTU |
| 站 | 1 | 1 |
| 接收流控制 | 无 | 无 |

| | | |
|---|---|---|
| 发送流控制 | 无 | 无 |
| 串行超时 | 60s | 60s |
| 类型 | RS-232 | RS-232 |

## 4.3 调试串行通讯

下面是引起通讯错误常见原因：

- 控制器与外部设备需使用相同的通讯参数。
- 若部分字符不能正确发送，则设备之间的奇偶校验或停止位不相匹配。

两个 RS-232 DTE 间直接用电缆连接，不需要 MODEM。此电缆将一个设备的发送数据输出连接到另一个设备的接收数据输入。控制器为 DTE 设备。

DTE 与 DCE 间用电缆连接。DTE 的发送数据输出与 DCE 的发送数据输入连接。MODEM 为 DCE 设备。

许多 RS-232 设备的某些管脚需要特殊信号电平。若信号达不到所需要求，通讯不能正常进行。例如：控制器中的 CTS 线必须在正确的电平上才能工作。若 CTS 为 OFF，则控制器不能发送。若 CTS 线未被连接，控制器将强制它达到正确值。若使用外设控制此线，应使它为 ON。

## 4.4 使用拨号 MODEM

此库函数提供了拨号 MODEM 的控制。用外置 MODEM 连接串口。

| | |
|---|---|
| modemlnit | 向拨号 MODEM 发送初始化字符串 |
| modemlnitstatus | 读取 MODEM 初始化运行状态 |
| modemlnitend | 结束 MODEM 初始化运行 |
| modemdial | 使用拨号 MODEM 连接外设 |
| modemdialstatus | 读取使用拨号 MODEM 连接外设的状态 |
| modemdialend | 中断使用拨号 MODEM 连接外设 |

通常使用 DTE~DCE 电缆将外置 MODEM 连接到 RS-232 口上。下面例举了 MODEM 的初始化和与远程拨号控制器的连接。

### 4.4.1 MODEM 初始化

```
#include “primitiv.h”
#include “dialup.h”
#include “string.h”
#include “serial.h”

void main(void)
{
    struct ModemInit initSettings;
    reserve_id portID;
```

```
    enum DialError status;
    enum DialState state;
    struct pconfig portSettings;

    portSettings.baud    = BAUD1200;
    portSettings.duples = FULL;
    portSettings.parity  = NONE;
    portSettings.data_bits     = DATA8;
    portSettings.stop_bits     = STOP1;
    portSettings.flow_rx       = DISABLE;
    portSettings.flow_rx       = DISABLE;
    portSettings.type     =RS232_MODEM;
    portSettings.timeout       600;
    request_resource(IO_SYSTEM);
    set_port(com1,&portSettings);
    release_resource(IO_SYSTEM);

    initSettings.port = com1;
    strcpy(initSettings.modemCommand, "F1Q0V1X1 S0=1");
    if (modemInit(&initSettings,&portID) == DE_NoError)
    {
        do
        {
            release_processor();
            modemInitStatus(com1,portID,&status,&state);
        }
        while (state == DS_Calling);

        modemInitEnd(com1,portID,&status);
    }
}
```

## 4.4.2 连接远程控制器

下面的代码所示为如何使用 MODEM 连接远程控制器。

```
#include “primitiv.h”
#include “dialup.h”
#include “string.h”
#include “serial.h”

void main(void)
{
    struct ModemSetup dialSettings;
    reserve_id portID;
```

```
    enum DialError status;
    enum DialState state;
    struct pconfig portSettings;

    portSettings.baud    = BAUD1200;
    portSettings.duples = FULL;
    portSettings.parity  = NONE;
    portSettings.data_bits     = DATA8;
    portSettings.stop_bits     = STOP1;
    portSettings.flow_rx       = DISABLE;
    portSettings.flow_rx       = DISABLE;
    portSettings.type     =RS232_MODEM;
    portSettings.timeout       600;
    request_resource(IO_SYSTEM);
    set_port(com1,&portSettings);
    release_resource(IO_SYSTEM);

    dialSettings.port                = com1;
    dialSettings.dialAttempts        = 3;
    dialSettings.detectTime          = 60;
    dialSettings.pauseTime           = 30;
    dialSettings.dialmethod          = 0;
    strcpy(dialSettings.modemCommand,"&F1 &A0 &K0 &M0 &B1");
    strcpy(dialSettings.phoneNumber,"555-1212");

    if (modemDial(&dialSettings,&portID) == DE_NoEorror)
    {
        do
        {
            release_processor();
            modemDialStatus(com1,portID,&status,&state);
        }
        while (state -- DS_Calling);
        if (state == DS_Connected)
        {
        }
        modemDialEnd(com1,portID,&status);
    }
}
```

# 第五章 实时操作系统运行机制

实时操作系统（RTOS）为编程人员提供了建立复杂应用的工具。RTOS 提供了一个驱动任务事件表，使得可以快速响应事件。内部任务通讯和事件通知函数可在任务之间传递信息。资源函数可更容易控制非共享资源。

## 5.1 任务管理

任务管理函数用于建立与结束任务。任务独立执行程序。

初始任务可以建立附加任务。RTOS 支持最多 16 个任务。任务执行时间表中有 4 个优先等级。

有 5 个 RTOS 函数用于任务管理：

create-task　　建立一个任务并准备运行。

end-task　　结束任务并释放资源

end-application　　结束所有应用程序型任务。通讯协议使用此函数在下载新的代码前终止应用程序。

installExitHandler　指定一个函数，当任务以 end-task 或 end-application function 结束时调用此函数。

getTaskinfo　　返回任务信息

## 5.2 资源管理

资源管理函数用于访问非共享资源。

有三个 RTOS 函数用于资源管理。

**request-resouce**　请求访问资源。若资源无效时进行等待。

**poll-resouce**　　请求访问资源。若资源无效时继续运行。

**release-resouce**　释放资源给其它任务使用。

RTOS 定义了三个系统资源，用于 I/O 驱动器元件、内存分配函数与通讯协议。

应用程序可定义所需的其它资源。注意不能复制任何 primitive.h 中作为系统资源的资源编号。宏 RTOS-RESOURCES 用于定义可用资源的编号。

### 5.2.1 I/O 系统资源

I/O 系统控制对所有使用 I/O 系统的函数的访问。I/O 系统可被 C 应用程序、梯形逻辑程序、通讯协议和后台 I/O 操作共享。对资源的控制不能超过 0.1 秒，否则后台运行将不能正常运行。

I/O 系统资源必须在使用下面任何一种函数前得到。

| 函数 | 定义 |
|---|---|
| addregassignment | 添加 I/O 模块到寄存器分配中 |
| alarmin | 返回绝对的报警时间 |

| | |
|---|---|
| auto-pid | 设置模块以指定速度自动运行 |
| clearallfotcing | 清除所有强制 |
| clear-error | 清除串口错误计数 |
| clear-Pid | 设置所有模块变量为 0 |
| clear-protocol-stasus | 清除协议信息与错误计数 |
| clear-regassignment | 清除寄存器分配 |
| clear-translationtable | 清除寄存器分配 |
| clearsftranslation | 清除存储与转发译码表入口 |
| counter | 读取计数器输入模块 |
| dbase | 从 I/O 数据库读取值 |
| getclock | 读取实时时钟 |
| getclockalarm | 读取实时时钟报警设置 |
| get-pid | 返回 PID 模块变量值 |
| master-message | 发送协议信息到另一个设备 |
| overrideDbase | 写入值到强制 I/O 数据库中 |
| resetclockalarm | 复位实时时钟，使它可在第二天相同时间复现 |
| setclock | 设置实时时钟 |
| setclockalarm | 设置实时时钟报警 |
| setdbase | 写值到 I/O 数据库中 |
| setforceflag | 为寄存器设置强制标记状态 |
| set-pid | 将 PID 模块变量设置为指定值 |
| set-port | 设置串口通讯参数 |
| set-protocol | 设置协议参数并运行协议 |
| setsftranslation | 写存储和转发译码表入口地址 |
| sleep | 调用休眠模式 |

### 5.2.2 动态内存资源

动态内存资源控制所有内存资源分配函数。这些函数从系统堆栈分配内存。堆栈被所有任务共享。

使用下面任一种函数前需得到动态内存资源。

| 函数 | 定义 |
|---|---|
| calloc | 动态分配数据空间 |
| free | 释放动态分配内存 |
| malloc | 动态分配数据空间 |
| realloc | 改变动态分配空间的大小 |

### 5.2.3 MODBUS-PARSER 资源

此资源用于 MODBUS 通讯协议驱动程序，对通用信息的访问进行分配。

### 5.2.4 预防优先权倒置

当较高优先权任务 TASK H 需要的资源已被较低优先权任务 TASK L 获得时，高优先权任务被中断，直到 TASK L 释放资源。这叫做优先权倒置。

为防止优先权倒置现象的发生，可执行继承优先权。如上例所述，低优先级任务 TASK L 被提升为 TASK H 的优先级，直到其释放所需的资源。此时，TASK L 返回到最初优先级。TASK H 便可得到此资源。

注意这不能避免一个任务需要资源，另一个任务已得到此资源时发生死锁。死锁是应用程序的设计错误。

## 5.3 内部任务通讯

内部任务通讯函数传递任务间的信息。可用于数据交换和任务同步化。信息排队等待，直到接收任务准备处理数据。

有 5 个 RTOS 函数用于中间任务通讯。

| | |
|---|---|
| Send-message | 发送信息包到另一个任务 |
| receive-message | 从任务信息队列读取接收信息或当队列为空时进行等待。 |
| Poll-message | 从任务信息队列中读取接收信息。若队列为空继续执行任务。 |
| Allocate-envelope | 从被 RTOS 维护的自由池中获得信息邮箱。 |
| Deallocate-envelope | 返回信息到 RTOS 维护的自由池中。 |

## 5.4 事件通知

有四个 RTOS 函数用于事件通知：

| | |
|---|---|
| wait-event | 等待事件发生 |
| poll-event | 检查是否有事件发生。若无事件发生，则继续执行。 |
| Signal-event | 发送信号表示某一事件发生。 |
| Interrupt-signal-event | 为中断处理器中发生的事件发送信号。此函数只能从中断处理器调用。 |

有两个支持函数，它们不是 RTOS 中的一部分，但可用于事件：

| | |
|---|---|
| startTimedEvent | 以规则的间隔使能事件的信号 |
| endTimedEvent | 结束规则事件的信号 |

RTOS 为通讯口管理和后台运行定义事件。应用程序可定义其它所需事件。

### 5.4.1 系统事件

下表为 RTOS 定义的事件

| 事件 | 定义 |
|---|---|
| 后台 | 此事件触发后台 I/O 程序的运行。应用程序不能使用此事件。 |
| COM1-FREE | 此事件用于 COM1 口的串行超时程序中。应用程序不能使用此事件。 |
| COM1-RCVR | 此事件用于通讯协议中，表明在 COM1 上接收到字符和信息。 |

| | |
|---|---|
| COM2-FREE | 此事件用于 COM2 口的串行超时程序中。应用程序不能使用此事件 |
| COM2-RCVR | 此事件用于通讯协议中，表明在 COM2 上接收到字符和信息。 |
| COM3-RCVR | 此事件用于通讯协议中，表明在 COM3 上接收到字符和信息。 |
| COM4-RCVR | 此事件用于通讯协议中，表明在 COM4 上接收到字符和信息。 |
| NEVER | 此事件永远不会发生。可以通过使某一任务等待该 NEVER 任务从而禁止任务。 |

## 5.5 错误报告

错误报告函数用于可共享的 I/O 驱动程序，将错误信息返回到调用任务中。这些函数可确保由一个任务产生的错误代码不会在另一个任务中报告。某些函数使用的 error 变量可在当前任务读取前被另一个任务更改。

有两个 RTOS 函数用于错误报告。

**checker-error**　检查当前任务的错误代码

**report-error**　设置当前任务代码

# 第六章专用控制 C 函数库

EMC 提供一个面向过程控制的运行函数库。这些函数是对 EMC 编译器运行库的补充。

这一章的第一部分简要描述 EMC 函数集。第二部分描述 EMC 头文件的内容。最后一部分详细描述运行库函数。

## 6.1 函数集

下面的部分简要描述 EMC 运行库所包含的函数。

### 6.1.1 校验和函数

在 **csum.h** 文件中定义了下面这些和校验和相关的函数。

| 函数 | 定义 |
|---|---|
| checksum | 计算加法、CRC-16、CRC-CCITT 和异或校验和。 |
| crc_revers | 使用保留的 CRC 算法进行 CRC 校验和计算。 |

### 6.1.2 系统控制函数

在 **system.h** 文件中定义了下面这些与系统初始化和获取系统信息相关的函数。其中的一些函数的用法可以在 **appstart.c** 中找到，这些函数在应用程序中有特定的应用。

| 函数 | 定义 |
|---|---|
| applicationChecksum | 返回应用程序的校验和 |
| endTimedEvent | 终止由 startTimedEvent 指定的事件产生的信号 |
| getBootType | 获得启动方式 |
| getProgramStatus | 返回应用程序的执行状态 |
| getVersion | 返回固件版本信息 |
| getWakeSource | 获得唤醒资源 |
| setBootType | 设置启动方式 |
| setProgramStatus | 设置应用程序的执行状态 |
| setWakeSource | 设置唤醒资源 |
| sleep | 使控制器处于睡眠模式 |
| startTimedEvent | 指定一个事件发生在有规则的时间间隔 |
| startup_task | 返回系统启动例程的地址 |
| system_start | 缺省的启动例程 |

### 6.1.3 实时操作系统函数

在 **primitiv.h** 文件中定义了下面这些和实时操作系统(real time operating system----RTOS)相关的函数。

| 函数 | 定义 |
|---|---|
| allocate_envelope | 向 RTOS 请求一个消息邮箱 |
| check_error | 检查当前任务的错误码 |
| create_task | 创建一个新任务，并且使新任务处于准备执行的状态 |
| deallocate_envelope | 返回一个消息邮箱到 RTOS |

| | |
|---|---|
| end_application | 结束所有的 APPLICATION 类型的任务。 |
| end_task | 结束一个任务 |
| GetTaskInfo | 返回一个任务的信息 |
| InstallExitHandler | 指定一个函数。当一个任务结束时，运行这个函数。 |
| interrupt_signal_event | 用信号报告一个中断句柄产生的事件 |
| poll_event | 检查一个事件的发生 |
| poll_message | 检查来自另一个任务的消息 |
| poll_resource | 检查非共享资源的可用性 |
| receive_message | 接收来自另一个任务的消息 |
| release_processor | 释放对处理器的控制，使其它任务可以使用处理器 |
| release_resource | 释放非共享资源 |
| report_error | 报告针对当前任务的错误码 |
| request_resource | 向操作系统请求一个非共享资源 |
| send_message | 向另一个任务发送一个消息 |
| signal_event | 用信号报告一个事件的发生 |
| wait_event | 等待一个事件的发生 |

### 6.1.4 实时时钟函数

在 **rtc.h** 文件中定义了下面这些和实时时钟相关的函数。

| 函数 | 定义 |
|---|---|
| alarmIn | 根据给定的时间长度，返回报警的绝对时间 |
| getclock | 读取实时时钟 |
| getClockAlarm | 读取实时时钟报警设置 |
| installClockHandler | 安装一个实时时钟报警的句柄 |
| jiffy | 读取系统精确绝对时钟 |
| resetClockAlarm | 重设实时时钟报警，可以使报警在下一天的同一时间发生 |
| setclock | 设置实时时钟 |
| setClockAlarm | 设置实时时钟报警 |
| setjiffy | 设置系统精确绝对时钟 |

### 6.1.5 串口通信函数

在 **serial.h** 文件中定义了下面这些和串口通信相关的函数。其余的串口通信函数在 EMC 运行库中定义。

| 函数 | 定义 |
|---|---|
| clear_errors | 清除串口错误计数器 |
| clear_tx | 清除串口发送缓冲区 |
| getPortCharacteristics | 读取关于串口支持的特征的信息 |
| get_port | 读取串口通信参数 |
| get_status | 读取串口状态和错误计数器 |
| install_handler | 安装串口字符接收句柄 |
| portConfiguration | 获得串口设置表的指针 |
| portIndex | 获得串口索引 |

| | |
|---|---|
| portStream | 由串口索引获得串口 |
| queue_mode | 设置串口发送模式 |
| route | 重定向标准 I/O 流 |
| setDTR | 控制 RS232 口 DTR 信号 |
| set_port | 设置串口通信参数 |

### 6.1.6 通信协议函数

在 **protocol.h** 文件中定义了下面这些和通信协议相关的函数。

| 函数 | 定义 |
|---|---|
| checkSFTranslationTable | 检查翻译表中的无效入口 |
| clear_protocol_status | 清除协议消息和错误计数器 |
| clearSFTranslationTable | 清除所有存储和传送（store and forward）翻译表的入口 |
| get_protocol | 读取协议参数 |
| get_protocol_status | 读取协议消息和错误计数器 |
| GetSFMapping | 这个函数已经不再使用，不再有任何功能 |
| getSFTranslation | 读取存储和传送（store and forward）翻译表 |
| master_message | 向另一个设备发送一个协议消息 |
| modbusExceptionStatus | 响应读异常状态函数 |
| modbusSlaveID | 响应读从站 ID 函数 |
| set_protocol | 设置协议参数并开始执行协议 |
| setSFMapping | 这个函数已经不再使用，不再有任何功能 |
| setSFTranslation | 写存储和传送（store and forward）翻译表 |
| start_protocol | 根据存储的参数，开始执行协议 |

### 6.1.7 拨号调制解调器函数

在 **dialup.h** 文件中定义了下面这些和拨号调制解调器相关的函数。

| 函数 | 定义 |
|---|---|
| modemInit | 向调制解调器发送初始化字符串 |
| modemInitStatus | 返回调制解调器初始化操作的状态 |
| modemInitEnd | 中止调制解调器初始化操作 |
| modemDial | 连接一个远程拨号控制器 |
| modemDialStatus | 返回一个远程拨号控制器的连接的状态 |
| modemDialEnd | 中止连接远程拨号控制器 |

### 6.1.8 I/O 数据库函数

在 **database.h** 文件中定义了下面这些和 I/O 数据库相关的函数。

| 函数 | 定义 |
|---|---|
| clearAllForcing | 清除全部强制 |
| dbase | 从 I/O 数据库中读取值 |
| getForceFlag | 获得一个寄存器的强制标志状态 |
| overrideDbase | 向被强制的 I/O 数据库中写入值 |
| setdbase | 向 I/O 数据库中写入值 |

| | |
|---|---|
| setForceFlag | 设置一个寄存器的强制标志状态 |

## 6.1.9 I/O 硬件函数

在 **iohw.h** 文件中定义了下面这些和操作硬件相关的函数。

| 函数 | 定义 |
|---|---|
| clearStatusBit | 清除控制器状态码的位 |
| forceLed | 设置强制 LED 的状态 |
| getStatusBit | 获得控制器状态码的位 |
| interruptInput | 读取输入中断的状态 |
| interval | 设置软件计时器的最小分辨的时间间隔 |
| setStatus | 设置控制器状态码 |
| setStatusBit | 设置控制器状态码的位 |
| settimer | 设置软件计时器，同时开始计数 |
| timer | 读软件计时器的内容 |
| wd_auto | 选择自动重新触发看门狗 |
| wd_manual | 选择手动重新触发看门狗 |
| wd_pulse | 重新触发看门狗 |

## 6.1.10 分配寄存器函数

在 **rassign.h** 文件中定义了下面这些和分配寄存器相关的函数。

| 函数 | 定义 |
|---|---|
| addRegAssignment | 向寄存器分配一个 I/O 模块 |
| clearRegAssignment | 清除寄存器分配 |
| getIOErrorIndication | 获得一个 I/O 模块错误状态的控制标志 |
| setOutputsInStopMode | 在梯形图停止模式下，获得输出状态的控制标志 |
| setIOErrorIndication | 设置一个 I/O 模块错误状态的控制标志 |
| setOutputsInStopMode | 在梯形图停止模式下，设置输出状态的控制标志 |

## 6.1.11 PID 控制器函数

在 **pid.h** 文件中定义了下面这些和 PID 控制器相关的函数。

| 函数 | 定义 |
|---|---|
| auto_pid | 设置一个以特定速率自动执行的块 |
| clear_pid | 设置全部块变量为 0 |
| get_pid | 返回一个块变量的值 |
| set_pid | 设置一个块变量为指定的值 |

## 6.2 头文件

头文件包含：运行库函数的声明、结构、宏。这些文件用**#include** 和引号包含。例如：

#include “iohw.h”

下面的部分描述每个头文件中的宏和结构。头文件按字母顺序列出。

# csum.h

在 **csum.h** 文件中定义了下面这些特定和校验类型的宏。

| 宏 | 定义 |
|---|---|
| ADDITIVE | 加法校验和 |
| BYTE_EOR | 字节方式异或校验和 |
| CRC_16 | CRC-16 类型 CRC 校验和（反转算法） |
| CRC_CCITT | CCITT 类型 CRC 校验和（反转算法） |

# database.h

在 **database.h** 文件中定义的宏：I/O 数据库寻址、数据库相关函数返回的错误码。

| 宏 | 定义 |
|---|---|
| AB | 指定 Allan-Bradley 数据库寻址 |
| DB_BADSIZE | 错误码：超出指定的地址范围 |
| DB_BADTYPE | 错误码：指定了错误的数据库寻址类型 |
| DB_OK | 错误码：没有错误发生 |
| LINEAR | 指定线性数据库寻址 |
| MODBUS | 指定 Modbus 数据库寻址 |
| NUMAB | 在 Allan-Bradley 数据库中的寄存器数量 |
| NUMCOIL | 在 Modbus 线圈部分的寄存器数量 |
| NUMHOLDING | 在 Modbus 保持寄存器部分的寄存器数量 |
| NUMINPUT | 在 Modbus 输入寄存器部分的寄存器数量 |
| NUMLINEAR | 在线性数据库中的寄存器数量 |
| NUMSTATUS | 在 Modbus 状态部分的寄存器数量 |
| START_COIL | 在线性数据库中线圈部分的起始地址 |
| START_HOLDING | 在线性数据库中保持寄存器部分的起始地址 |
| START_INPUT | 在线性数据库中输入寄存器部分的起始地址 |
| START_STATUS | 在线性数据库中状态部分的起始地址 |

# dialup.h

在 **dialup.h** 文件中定义了下列和应用程序相关的宏。

| 宏 | 定义 |
|---|---|
| MODEM_CMD_MAX_LEN | 调制解调器初始化命令字符串的最大长度 |
| PHONE_NUM_MAX_LEN | 电话号码字符串的最大长度 |

在 **dialup.h** 文件中定义两个枚举类型。DialError 枚举类型定义拨号调制解调器函数的错误响应。

| 值 | 定义 |
|---|---|
| DE_NoError | 没有错误发生。 |
| DE_BadConfig | 调制解调器配置结构包含错误 |
| DE_NoModem | 串口没有被设置为调制解调器（端口类型必须是 RS232_MODEM）。或者控制器串口没有和调制解调器连接。 |
| DE_InitError | 调制解调器初始化错误（调制解调器可能没有打开） |

| | |
|---|---|
| DE_NoDialTone | 调制解调器没有检测到拨号音，或者调制解调器的 S6 参数设定的时间太短。 |
| DE_BusyLine | 被叫电话号码正忙。 |
| DE_CallAborted | 正在进行的呼叫被用户终止。 |
| DE_FailedToConnect | 调制解调器不能和远端相连。 |
| DE_CarrierLost | 和远端的连接丢失（调制解调器报告没有载波）。载波消失的时间超过了调制解调器的 S10 参数的设置 |
| DE_NotInControl | 串口被另一个调制解调器函数使用，或者正在响应一个拨入。 |

DialState 枚举类型定义 ModemDial 操作的状态。

| 值 | 定义 |
|---|---|
| DS_Inactive | 调制解调器没有使用串口。 |
| DS_Calling | 控制器正在和远程控制器建立连接。 |
| DS_Connected | 控制器已经连接远程控制器。 |
| DS_Terminating | 控制器正在终止和远程控制器的连接。 |

在 **dialup.h** 文件中定义两个结构。ModemInit 结构详细说明了 modemInit 函数的调制解调器初始化参数。

```
struct ModemInit
{
    FILE *    port;
    char modemCommand[MODEM_CMD_MAX_LEN + 2];
};
```

- Port 是连接调制解调器的串口。
- ModemCommand 是对调制解调器初始化的字符串。当向调制解调器发送命令时，字符串”AT”将自动加在命令的前面,同时回车符号加在命令的后面。针对你使用的调制解调器的命令字符串，请参考附录 B：*调制解调器命令*。

ModemSetup 结构详细说明了 modemDial 函数用到的调制解调器初始化和拨号参数。

```
struct ModemSetup
{
    FILE *          port;
    unsigned short dialAttempts;
    unsigned short detectTime;
    unsigned short pauseTime;
    unsigned short dialmethod;
    char       modemCommand[MODEM_CMD_MAX_LEN + 2];
    char       phoneNumber[PHONE_NUM_MAX_LEN + 2];
};
```

- Port 是连接调制解调器的串口。
- dialAttempts 是控制器尝试连接远程控制器的最大次数，当达到尝试次数，还没有连接，将放弃并报告错误。

- detectTime 是控制器等待载波被检测到的秒数。从尝试拨号的时刻开始计时。
- pauseTime 是控制器两次尝试连接之间的时间间隔，单位是秒。
- dialmethod 设置脉冲或音频拨号。设置为 0 时音频拨号；设置为 1 时，表示脉冲拨号。
- modemCommand 是调制解调器使用的初始化串。当向调制解调器发送命令时，字符串”AT”将自动加在命令的前面,同时回车符号加在命令的后面。针对你使用的调制解调器的命令字符串，请参考附录 B：*调制解调器命令*。
- phoneNumber 是远程控制器的电话号码。当向调制解调器发送电话号码时，字符串”ATD”和拨号方法将自动加在命令的前面，同时回车符号加在命令的后面。

# iohw.h

**iohw.h** 文件定义了几个宏。

| 宏 | 定义 |
|---|---|
| TIMED_OUT | 指定的计时器已经到达零 |
| TIMER_BADINTERVAL | 错误码：无效的计时器计数间隔 |
| TIMER_BADTIMER | 错误码：无效的计时器 |
| TIMER_BADVALUE | 错误码：无效的计时器值 |
| TIMER_MAX | 最后的有效的软件计时器的数量。 |

在 **iohw.h** 文件中定义两个结构。

```
/*  LED 控制结构 */
struct ledControl_tag {
    unsigned state;
    unsigned time;
    };
```

- state 是缺省的 LED 的状态，它或者是 LED_ON，或者是 LED_OFF。
- time 是以分钟为单位的时间，经过这段时间后，LED 电源就被置为缺省的状态。

```
/*  计时器信息结构 */
struct timer_info {
    unsigned time;
    unsigned interval;
    unsigned interval_remaining;
    unsigned flags;
    unsigned duty_on;
    unsigned duty_period;
    unsigned channel;
    unsigned bit;
    };
```

- time 是计时器中剩余的时间，以滴答为单位。
- interval 是一个计时器滴答的时间长度。以 1/10 秒为单位
- interval_remaining。

- flags 计时器类型和状态位（NORMAL、PULSE_TRAIN、DUTY_CYCLE、TIMEOUT、TIMED_OUT），在任何时刻，可以有多个条件为真。
- duty_on 是方波输出的高电平部分的长度。它只被 **pulse** 函数用到。
- duty_period 是方波输出的时间长度。它只被 **pulse** 函数用到。
- channel 和 bit 指定数据输出指针。它被 **pulse,pulse_train** 和 **timeout** 函数用到。

# pid.h

**pid.h** 文件定义了关于操作 PID 块的宏。

| 宏 | 定义 |
|---|---|
| AO | 变量名：报警输出的地址 |
| CA | 变量名：多设置点的来源 |
| CR | 变量名：控制寄存器 |
| DB | 变量名：死区 |
| DO | 变量名：递减输出 |
| ER | 变量名：错误 |
| EX | 变量名：执行周期 |
| FS | 变量名：最大输出限制 |
| GA | 变量名：增益 |
| HI | 变量名：高报警设置点 |
| IB | 变量名：输入偏移 |
| IH | 变量名：禁止执行的地址 |
| IN | 变量名：积分误差 |
| IO | 变量名：递加输出 |
| IP | 变量名：输入源 |
| LO | 变量名：低报警设置点 |
| OB | 变量名：输出偏移 |
| OP | 变量名：输出 |
| PE | 变量名：时间间隔 |
| PID_ALARM | 控制寄存器掩码：报警使能 |
| PID_ALARM_ABS | 控制寄存器掩码：绝对报警 |
| PID_ALARM_ACK | 状态寄存器掩码：报警确认 |
| PID_ALARM_DEV | 控制寄存器掩码：偏移报警 |
| PID_ALARM_ONLY | 控制寄存器掩码：报警块 |
| PID_ALARM_RATE | 控制寄存器掩码：比率报警 |
| PID_ANALOG_IP | 控制寄存器掩码：模拟输入 |
| PID_ANALOG_OP | 控制寄存器掩码：模拟输出 |
| PID_BAD_BLOCK | 返回码：确定的坏块数 |
| PID_BAD_IO_IP | 状态寄存器掩码：块输入的 I/O 错误 |
| PID_BAD_IO_OP | 状态寄存器掩码：块输出的 I/O 错误 |
| PID_BLOCK_IP | 控制寄存器掩码：从另一个块的输出获得的输入 |
| PID_BLOCKS | PID 块的数量 |

| | |
|---|---|
| PID_CLAMP_FULL | 状态寄存器掩码：输出被固定在全程刻度 |
| PID_CLAMP_ZERO | 状态寄存器掩码：输出被固定在 0 刻度 |
| PID_ER_SQR | 控制寄存器掩码：错误值的平方根 |
| PID_HI_ALARM | 状态寄存器掩码：检测到高报警 |
| PID_INHIBIT | 状态寄存器掩码：外部禁止输入为开状态 |
| PID_LO_ALARM | 状态寄存器掩码：检测到低报警 |
| PID_MANUAL | 状态寄存器掩码：块处于手动模式 |
| PID_MODE_AUTO | 控制寄存器掩码：自动模式 |
| PID_MODE_MANUAL | 控制寄存器掩码：手动模式 |
| PID_MOTOR_OP | 控制寄存器掩码：马达脉冲连续输出 |
| PID_NO_ALARM | 控制寄存器掩码：报警禁止 |
| PID_NO_ER_SQR | 控制寄存器掩码：普通错误 |
| PID_NO_IP | 控制寄存器掩码：无输入（不是 IP） |
| PID_NO_OP | 控制寄存器掩码：无输出 |
| PID_NO_PV_SQR | 控制寄存器掩码：正常 PV |
| PID_NO_SP_TRACK | 控制寄存器掩码：设置点跟踪禁止 |
| PID_OK | 返回码：操作成功 |
| PID_OUT_DB | 状态寄存器掩码：PID 控制器超出死区 |
| PID_PID | 控制寄存器掩码：PID 控制块 |
| PID_PULSE_OP | 控制寄存器掩码：脉冲连续输出 |
| PID_PV_SQR | 控制寄存器掩码：PV 的平方根 |
| PID_RATE_CLAMP | 状态寄存器掩码：比率增益限制在最大值 |
| PID_RATIO_BIAS | 控制寄存器掩码：比例/偏移控制块 |
| PID_RUNNING | 状态寄存器掩码：块正在执行 |
| PID_SP_CASCADE | 控制寄存器掩码：多设置点 |
| PID_SP_NORMAL | 控制寄存器掩码：设置点保存在 SP |
| PID_SP_TRACK | 控制寄存器掩码：设置点跟踪使能 |
| PV | 变量名：过程值 |
| RA | 变量名：比率时间 |
| RE | 变量名：重置时间 |
| SP | 变量名：设置点 |
| SR | 变量名：状态寄存器 |
| ZE | 变量名：最小输出限制 |

# primitiv.h

**primitiv.h** 文件定义了关于 RTOS 函数的宏。

| 宏 | 定义 |
|---|---|
| APPLICATION | 指定一个应用类型的任务，所有应用类型的任务都能被一个结束应用函数终止。 |
| BACKGROUND | 系统事件：背景 I/O 请求。背景 I/O 任务在这个事件中使用。它不能用在一个应用程序中。 |
| COM1_FREE | 系统事件：com1 发送缓冲区没有满。这个事件由串口 I/O 驱动程序内部使用 |

| | |
|---|---|
| COM1_RCVR | 系统事件：com1 接收器处于活动状态。具体意义依赖于被安装的字符处理句柄 |
| COM2_FREE | 系统事件：com2 发送缓冲区没有满。这个事件由串口 I/O 驱动程序内部使用 |
| COM2_RCVR | 系统事件：com2 接收器处于活动状态。具体意义依赖于被安装的字符处理句柄 |
| COM3_RCVR | 系统事件：com3 接收器处于活动状态。具体意义依赖于被安装的字符处理句柄 |
| COM4_RCVR | 系统事件：com4 接收器处于活动状态。具体意义依赖于被安装的字符处理句柄 |
| DYNAMIC_MEMORY | 系统资源：全部内存分配函数，例如：malloc、alloc 和 zalloc |
| IO_SYSTEM | 系统资源：全部 I/O 硬件 |
| MAX_PRIORITY | 最大任务优先级 |
| MODBUS_PARSER | 系统资源：Modbus 协议消息分析器 |
| MODEM_MSG | 系统事件：生成一个新调制解调器消息 |
| MSG_DATA | 指定一个邮箱中的数据区包含一个数值 |
| MSG_POINTER | 指定一个邮箱中的数据区包含一个指针 |
| NEVER | 系统事件：这个时间将永远不发生 |
| NO_ERROR | 错误码：没有错误发生 |
| RTOS_ENVELOPES | RTOS 邮箱的数量 |
| RTOS_EVENTS | RTOS 事件的数量 |
| RTOS_PRIORITIES | RTOS 任务优先级的数量 |
| RTOS_RESOURCES | RTOS 资源标志的数量 |
| RTOS_TASKS | RTOS 任务的数量 |
| STACK_SIZE | 机器堆栈的大小 |
| SYSTEM | 指定一个系统类型的任务。系统任务不能被一个结束应用函数终止。 |
| TS_EXECUTING | 任务状态：任务正在执行 |
| TS_READY | 任务状态：任务已准备好 |
| TS_WAIT_RESOURCE | 任务状态：任务被锁定，正在等待一个资源 |
| TS_WAIT_ENVELOPE | 任务状态：任务被锁定，正在等待一个邮箱 |
| TS_WAIT_EVENT | 任务状态：任务被锁定，正在等待一个事件 |
| TS_WAIT_MESSAGE | 任务状态：任务被锁定，正在等待一个消息 |

在 **primitiv.h** 文件中定义两个结构。

```
    /*  消息邮箱结构*/
typedef struct env {
    struct env      *link;
    unsigned        source;
    unsigned        destination;
    unsigned        type;
    unsigned long data;
    unsigned        owner;
    }
```

envelope;

- link 是邮箱队列的下一个指针。这个域在 RTOS 中使用。应用程序中没有用到它。
- source 是发送消息的任务的任务 ID。它由 **send_message** 函数自动定义。接收任务通过读取这个域来确定消息的来源。
- destination 是接收消息的任务的任务 ID。它必须在调用 **send_message** 函数之前被定义。
- type 指定在数据域中的数据类型。它可以是 **MSG_DATA、MSG_POINTER，**或者由应用程序定义的任何值。这个域不是必须要求的
- data 是消息数据。这个域可能包含一个数据或者一个指针。这个域的使用由应用程序决定。
- owner 是拥有这个邮箱的任务。这个由 RTOS 设置，并且不能被应用程序改变。

```
/* 任务信息结构 */
typedef struct taskInformation_tag {
    unsigned taskID;
    unsigned priority;
    unsigned status;
    unsigned requirement;
    unsigned error;
    unsigned type;
    } TASKINFO;
```

- taskID 是任务的标识符。
- priority 是任务的运行优先权。
- status 是任务的当前运行状态。可以是下列值之一：**T**S_READY、TS_EXECUTING、TS_WAIT_ENVELOPE 、 TS_WAIT_EVENT 、 TS_WAIT_MESSAGE 、TS_WAIT_RESOURCE。参见上面的定义。
- 当任务正在等待一个事件或者资源，requirement 将被使用。如果 status 域是 TS_WAIT_EVENT，requirement 表示它正在等待的那个事件。如果 status 域是 TS_WAIT_RESOURCE，requirement 表示它正在等待的那个资源。
- error 是任务错误码。它和 check_error 函数的返回值相同。
- type 是任务类型。它可以是 SYSTEM 或 APPLICATION。

# protocol.h

**protocol.h** 文件定义了关于特定的通信协议参数的宏。

| 宏 | 定义 |
|---|---|
| FORCE_MULTIPLE_COILS | Modbus 函数码 |
| FORCE_SINGLE_COIL | Modbus 函数码 |
| LOAD_MULTIPLE_REGISTERS | Modbus 函数码 |

| | |
|---|---|
| LOAD_SINGLE_REGISTER | Modbus 函数码 |
| MM_BAD_ADDRESS | 主消息状态：无效的数据库地址 |
| MM_BAD_FUNCTION | 主消息状态：无效的函数码 |
| MM_BAD_LENGTH | 主消息状态：无效的消息长度 |
| MM_BAD_SLAVE | 主消息状态：无效的从站地址 |
| MM_NO_MESSAGE | 主消息状态：没有消息被发送 |
| MM_PROTOCOL_NOT_SUPPORTED | 主消息状态：选择的协议不被支持 |
| MM_RECEIVED | 主消息状态：响应被接收 |
| MM_SENT | 主消息状态：消息被发送 |
| MODBUS_ASCII | 指定在串口中使用 MODBUS_ASCII 协议 |
| MODBUS_RTU | 指定在串口中使用 MODBUS_RTU 协议 |
| NO_PROTOCOL | 指定在串口中不使用协议 |
| READ_COIL_STATUS | Modbus 函数码 |
| READ_EXCEPTION_STATUS | Modbus 函数码 |
| READ_HOLDING_REGISTER | Modbus 函数码 |
| READ_INPUT_REGISTER | Modbus 函数码 |
| READ_INPUT_STATUS | Modbus 函数码 |
| REPORT_SLAVE_ID | Modbus 函数码 |
| SF_ALREADY_DEFINED | 结果码：翻译已经在表中定义 |
| SF_INDEX_OUT_OF_RANGE | 结果码：无效的翻译表索引 |
| SF_NO_TRANSLATION | 结果码：入口没有定义一个翻译 |
| SF_PORT_OUT_OF_RANGE | 结果码：串口无效 |
| SF_STATION_OUT_OF_RANGE | 结果码：站地址无效 |
| SF_TABLE_SIZE | 在保存和发送表中的入口数 |
| SF_VALID | 结果码：翻译无效 |

在 **protocol.h** 文件中定义四个结构。

```
    /*  协议状态信息结构  */
struct prot_status {
    unsigned command_errors;
    unsigned format_errors;
    unsigned checksum_errors;
    unsigned cmd_received;
    unsigned cmd_sent;
    unsigned rsp_received;
    unsigned rsp_sent;
    unsigned command;
    int          task_id;
    unsigned stored_messages;
    unsigned forwarded_messages;
    };
```

- command_errors 是收到的无效命令码的消息数。

- format_errors 是收到的坏消息数据的消息数。
- checksum_errors 是收到的坏校验和的消息数。
- cmd_received 是收到的命令数。
- cmd_sent 是由 **master_message** 函数发送的命令数。
- rsp_received 是由 **master_message** 函数收到响应数。
- rsp_sent 是发送的响应数。
- command 是最后一个发送的协议命令的状态。
- task_id 是协议任务的 ID。这个域由 **set_protocol** 任务用来控制协议的执行。
- stored_messages 是保存的消息数，用来发送。
- forwarded_messages 是已经被发送的消息数。

```
/* 协议设置结构 */
struct prot_settings {
    unsigned char type;
    unsigned char station;
    unsigned char priority;
    unsigned SFMessaging;
    };
```

- type 是协议类型。它可以是下列值之一：NO_PROTOCOL、MODBUS_RTU、MODBUS_ASCII。
- station 是控制器的站地址。注意：每个串口都可以有一个不同的地址。有效的地址由协议规定。如果协议类型是 NO_PROTOCOL，这个域就不被使用。
- priority 是协议任务的任务优先级。如果协议类型是 NO_PROTOCOL，这个域就不被使用。
- SFMessaging 是保存和发送控制的使能标志。

```
/* 保存和发送消息翻译 */
struct SFTranslation {
    unsigned portA;
    unsigned stationA;
    unsigned portB;
    unsigned stationB;
    };
```

- portA 是第一个串口的索引号。这个索引号通过使用 portIndex 函数获得。
- stationA 是第一个站的站地址。
- portB 是第二个串口的索引号。这个索引号通过使用 portIndex 函数获得。
- portB 是第二个站的站地址。

```
/* 保存和发送状态码 */
struct SFTranslationStatus {
    unsigned index;
    unsigned code;
    };
```

- index 是状态码使用的保存和传送表的位置。

code 是状态码。它是下列值之一：SF_VALID、SF_INDEX_OUT_OF_RANGE、SF_NO_TRANSLATION、SF_PORT_OUT_OF_RANGE、SF_STATION_OUT_OF_RANGE、SF_ALREADY_DEFINED。

# rassign.h

在 protocol.h 文件中定义枚举类型---- ioModules。它包含下列 I/O 模块类型的符号常量。关于这些常量的用法，请参考*函数详细说明*中的函数 addRegAssignment。

AIN_E001_Temperature
AIN_E001_RAMBattery
AIN_E101
AOUT_E105
AOUT_E106
CNFG_clearPortCounters
CNFG_clearProtocolCounters
CNFG_LEDPower
CNFG_PIDBlock
CNFG_portSettings
CNFG_protocolSettings
CNFG_realTimeClock
CNFG_storeAndForward
CNTR_E001_CounterInputs
CNTR_E107
DIAG_commStatus
DIAG_controllerStatus
DIAG_forceLED
DIAG_protocolStatus
DIN_E001_DigitalInputs
DIN_E001_InterruptInput
DIN_E001_OptionSwitches
DIN_E102
DOUT_E104
DOUT_E109

# rtc.h

**rtc.h** 文件定义了关于实时时钟报警的宏。

| 宏 | 定义 |
|---|---|
| AT_NONE | 指定一个日报警的固定时间 |
| AT_ABSOLUTE | 禁用报警 |

在 **rtc.h** 文件中定义两个结构。

```
/* 实时时钟结构 */
struct clock {
    unsigned year;
    unsigned month;
    unsigned day;
    unsigned dayofweek;
    unsigned hour;
    unsigned minute;
```

```
    unsigned second;
    } ;
```

- year 是当前年。是 2 位十进制数字，范围是 00 到 99。
- month 是当前月。范围是 1 到 12。
- day 是当前日。范围是 1 到 31。
- dayofweek 是当前日的星期。范围是 1 到 7。由应用程序定义这个域的意义。
- hour 是当前小时。范围是 00 到 23。
- minute 是当前分钟。范围是 00 到 59。
- second 是当前秒。范围是 00 到 59。

```
/*  报警设置结构*/
typedef struct alarmSetting_tag {
    unsigned type;
    unsigned hour;
    unsigned minute;
    unsigned second;
    } ALARM_SETTING;
```

- type 指定报警类型。它可以是 AT_NONE 或 AT_ABSOLUTE。
- hour 指定报警发生的小时。
- minute 指定报警发生的分钟。
- second 指定报警发生的秒。

# serial.h

**serial.h** 文件定义了详细的串口通信参数的宏。

| 宏 | 定义 |
|---|---|
| BAUD110 | 指定 110 波特的端口速率 |
| BAUD115200 | 指定 115200 波特的端口速率 |
| BAUD1200 | 指定 1200 波特的端口速率 |
| BAUD150 | 指定 150 波特的端口速率 |
| BAUD19200 | 指定 19200 波特的端口速率 |
| BAUD2400 | 指定 2400 波特的端口速率 |
| BAUD300 | 指定 300 波特的端口速率 |
| BAUD38400 | 指定 38400 波特的端口速率 |
| BAUD4800 | 指定 4800 波特的端口速率 |
| BAUD57600 | 指定 57600 波特的端口速率 |
| BAUD600 | 指定 600 波特的端口速率 |
| BAUD75 | 指定 75 波特的端口速率 |
| BAUD9600 | 指定 9600 波特的端口速率 |

| | |
|---|---|
| com1 | com1 串口的文件对象的指针 |
| com2 | com2 串口的文件对象的指针 |
| com3 | com3 串口的文件对象的指针 |
| com4 | com4 串口的文件对象的指针 |
| DATA7 | 指定 7 比特字长 |
| DATA8 | 指定 8 比特字长 |
| DISABLE | 指定流控制为禁用 |
| ENABLE | 指定流控制为可用 |
| EVEN | 指定偶校验 |
| FOPEN_MAX | 重定义 **stdio.h** 中的宏 |
| FULL | 指定全双工 |
| HALF | 指定半双工 |
| NONE | 指定无奇偶校验 |
| NOTYPE | 指定串口类型为未知 |
| ODD | 指定奇校验 |
| RS232 | 指定串口是一个 RS-232 口 |
| RS232_MODEM | 指定串口是一个 RS-232 拨号调制解调器 |
| RS485_4WIRE | 指定串口是一个 4 线 RS-485 口 |
| SERIAL_PORTS | 串口数 |
| SIGNAL_CTS | I/O 线位掩码：清除并准备发送信号 |
| SIGNAL_DCD | I/O 线位掩码：载波检测信号 |
| SIGNAL_OFF | 指定一个信号被撤消 |
| SIGNAL_OH | I/O 线位掩码：挂机信号 |
| SIGNAL_ON | 指定一个信号被声称 |
| SIGNAL_RING | I/O 线位掩码：响铃信号 |
| SIGNAL_VOICE | I/O 线位掩码：声音/数字开关信号 |
| STOP1 | 指定一个停止位 |
| STOP2 | 指定两个停止位 |

在 **serial.h** 文件中定义下列结构：

```
/* 串口配置结构 */
struct pconfig {
    unsigned baud;
    unsigned duplex;
    unsigned parity;
    unsigned data_bits;
    unsigned stop_bits;
    unsigned flow_rx;
    unsigned flow_tx;
    unsigned type;
    unsigned timeout;
    };
```

- baud 是通信速率。它是 **BAUDxxx** 宏之一。
- duplex 是 **FULL** 或者 **HALF**。
- parity 是 **NONE**、**EVEN** 或者 **ODD** 之一。
- data_bits 是字长度。它是 **DATA7** 或者 **DATA8** 之一。
- stop_bits 是发送的停止位的数值。它是 **STOP1** 或者 **STOP2** 之一。
- flow_rx 指定接收器的流控制。它是 **DISABLE** 或者 **ENABLE** 之一。

  对于 com1 和 com2，当设置这个参数时，可以选择 XON/XOFF 流控制。它可以为可用和禁用。

  对于 com3 和 com4，当设置这个参数时，可以选择在接收消息后，接收器禁用。 它只能用在 Modbus RTU 协议。如果使用 Modbus RTU 协议，设置 flow_rx 为 **ENABLE**。否则，设置 flow_rx 为 **DISABLE**。。
- flow_tx 指定发送器的流控制。它是 DISABLE 或者 ENABLE 之一。

  对于 com1 和 com2，当设置这个参数时，可以选择 XON/XOFF 流控制。它可以为可用和禁用。

  对于 com3 和 com4，如果端口应该忽略 CTS 信号，就设置这个参数。设置这个参数为 ENABLE，端口将忽略 CTS 信号。

  如果端口没有使用 Modbus ASCII 协议，而是其它协议，你必须设置 flow_rx 为 DISABLE。如果使用 Modbus ASCII 协议或者没有使用协议，你可以设置 flow_rx 为 ENABLE 或者 DISABLE。在多数情况下，推荐使用 DISABLE。
- type 指定串口类型。它是 NOTYPE 或者 RS232 之一。
- timeout 指定驱动程序将等待的时间。也就是当传送缓冲区已满时，在它清楚缓冲区之前的这段时间。

```
/* 串口状态结构 */
struct pstatus {
    unsigned framing;
    unsigned parity;
    unsigned c_overrun;
    unsigned b_overrun;
    unsigned rx_buffer_size;
    unsigned rx_buffer_used;
    unsigned tx_buffer_size;
    unsigned tx_buffer_used;
    unsigned io_lines;
    };
```

- framing 是含有帧错误的接收字符的数量。
- parity 是含有奇偶校验错误的接收字符的数量。
- c_overrun 是含有超时错误的接收字符的数量。
- b_overrun 是接收缓冲区溢出错误的数量。

- rx_buffer_size 是接收缓冲区大小字节数。
- rx_buffer_used 是在接收缓冲区中的字节数。
- tx_buffer_size 是发送缓冲区大小字节数
- tx_buffer_used 是在发送缓冲区中的字节数。
- io_lines 是一个串口上的指示 I/O 线的状态的位标志域。这些线的值在不同端口之间不相同（参考下表）。这些信号可以通过和下列宏与运算来获得。

| 宏 | 支持 |
|---|---|
| SIGNAL_CTS | 和 CTS 输入状态匹配 |
| SIGNAL_DCD | 和 DCD 输入状态匹配 |
| SIGNAL_OH | 不支持—强制低位（1） |
| SIGNAL_RING | 不支持—强制低位（0） |
| SIGNAL_VOICE | 不支持—强制低位（0） |

```
/* 端口特性结构 */
typedef struct portCharacteristics_tag {
    unsigned dataflow;
    unsigned buffering;
    unsigned protocol;
    unsigned long options;
    } PORT_CHARACTERISTICS;
```

- dataflow 是串口支持的数据流选项的位标志。这些选项可以通过和下列宏与运算来获得。

| 宏 | 支持 |
|---|---|
| PC_FLOW_RX_RECEIVE_STOP | 当接收一个消息后，接收器禁用。 |
| PC_FLOW_RX_XON_XOFF | 接收器 Xon/Xoff 流控制 |
| PC_FLOW_TX_IGNORE_CTS | 发送器流控制忽略 CTS 信号 |
| PC_FLOW_TX_XON_XOFF | 发送器 Xon/Xoff 流控制 |

- buffering 描述被支持的缓冲区选项。目前没有被支持的缓冲区选项。
- protocol 描述被支持的协议项。下表中列出的宏是唯一被支持的。

| 宏 | 支持 |
|---|---|
| PC_PROTOCOL_RTU_FRAMING | Modbus RTU 桢 |

- options 描述增补的支持选项。目前没有增补的支持选项。

# system.h

在 **system.h** 文件中定义了状态信息和控制的宏。

| 宏 | 定义 |
|---|---|
| COLD_BOOT | 当 CPU 被重置时，冷启动开关按下 |
| CPU_CLOCK_RATE | 系统时钟频率，单位是：周期/每秒 |
| NEW_PROGRAM | 应用程序刚刚被加载 |
| PROGRAM_EXECUTED | 应用程序已经运行 |
| RUN | 运行/服务开关在运行位置 |

| | |
|---|---|
| SERVICE | 运行/服务开关在服务位置 |
| SLEEP_MODE_SUPPORTED | 支持睡眠功能 |
| STARTUP_APPLICATION | 指定应用开始时的任务 |
| STARTUP_SYSTEM | 指定系统开始时的任务 |
| VI_DATE_SIZE | 版本信息的时间域的字符数 |
| WS_ALL | 全部唤醒资源可用 |
| WS_COUNTER_0_OVERFLOW | 使计数器 0 作为唤醒资源的位掩码 |
| WS_COUNTER_1_OVERFLOW | 使计数器 1 作为唤醒资源的位掩码 |
| WS_COUNTER_2_OVERFLOW | 使计数器 2 作为唤醒资源的位掩码 |
| WS_INTERRUPT_INPUT | 使中断输入作为唤醒资源的位掩码 |
| WS_LED_POWER_SWITCH | 使 LED 电源开关作为唤醒资源的位掩码 |
| WS_NONE | 没有唤醒资源 |
| WS_REAL_TIME_CLOCK | 使实时时钟作为唤醒资源的位掩码 |
| WS_UNDEFINED | 未定义的唤醒资源 |

**system.h** 文件定义了用于 startup_task 函数的结构。

```
struct taskInfo_tag {
    void *address;
    unsigned stack;
    unsigned identity;
    };
```

- address 是指向启动例程的指针。
- stack 是为例程需要的堆栈的大小。
- identity 是创建的例程的类型（**STARTUP_APPLICATION** 或 **STARTUP_SYSTEM**）。

**system.h** 文件定义了用于 get_Version 函数的结构。

```
typedef struct versionInfo_tag {
    unsigned version;
    unsigned controller;
    char date[VI_DATE_SIZE + 1];
    char copyright[VI_STRING_SIZE + 1];
    } VERSION;
```

- version 是硬件的版本号。
- controller 是针对硬件的目标控制器。
- date 是一个包含硬件生成日期的信息的字符串。
- copyright 是一个包含安控科技版权信息的字符串。

## 6.3 函数详细说明

控制器 C 函数的详细说明按下面的形式组织。函数按字母循序排列。

**函数名** 说明开始于一个函数名称和一个简要描述

**语法** 语法给出函数的原型，指出返回类型和参数类型。列出所有必须的头文件。

**说明** 这里定义函数的调用参数和它的返回值。

**注** 这个部分包含函数的附加信息和使用中的注意事项。

**参考** 这个部分列出相关函数。

**例子** 例子给出函数用法的一个简要的样例。

# addRegAssignment

## 增加寄存器分配

**语法**

```
#include “rassign.h”
BOOLEAN addRegAssignment(
    unsigned    moduleType,
    unsigned    moduleAddress,
    unsigned startingRegister1,
    unsigned startingRegister2,
    unsigned startingRegister3,
    unsigned startingRegister4
    );
```

**说明**

**addRegAssignment** 函数增加一个 I/O 模块到当前的 moduleType 类型的寄存器分配表中。下列符号常量是有效的 moduleType 值：

| | |
|---|---|
| AIN_E001_Temperature | CNTR_E001_CounterInputs |
| AIN_E001_RAMBattery | CNTR_E107 |
| AIN_E101 | DIAG_commStatus |
| AOUT_E105 | DIAG_controllerStatus |
| AOUT_E106 | DIAG_forceLED |
| CNFG_clearPortCounters | DIAG_protocolStatus |
| CNFG_clearProtocolCounters | DIN_E001_DigitalInputs |
| CNFG_LEDPower | DIN_E001_InterruptInput |
| CNFG_PIDBlock | DIN_E001_OptionSwitches |
| CNFG_portSettings | DIN_E102 |
| CNFG_protocolSettings | DOUT_E104 |
| CNFG_realTimeClock | DOUT_E109 |
| CNFG_storeAndForward | |

moduleAddress 为模块指定一个唯一的地址。设置正确的地址的方法，请参考《ELadder 使用手册》中*寄存器的分配*的章节。模块地址 com1、com2、com3 或 com4，分别指定参数 moduleAddress 为 0、1、2、3。有的模块类型没有地址，例如：CNFG_LEDPower，指定 moduleAddress 为-1。

startingRegister1 指定任何没有使用的连续寄存器块的第一个地址。请参考在*寄存器分配参考*中列出的模块对寄存器块的类型和数量的要求。对模块的数据读写保存在这个寄存器区。

如果指定的模块类型有多种类型的 I/O，使用参数 startingRegister2、startingRegister3 和 startingRegister4。每个开始寄存器为模块中每个输入和输出指定没有使用的连续寄存器块的第一个地址。请参考在*寄存器分配参考*中列出的模块关于模块 I/O 类型的描述。如果没有多种类型的 I/O，请将参数 startingRegister2、startingRegister3 和 startingRegister4 指定为 0。

**注**

当模块加到寄存器分配时，有些 CNFG 类型的模块（例如：CNFG_realTimeClock），指定的寄存器将用当前的参数初始化。

在使用 addRegAssignment 函数进行寄存器分配之前，请先调用 clearRegAssignment。

这个函数不检查重复或重叠的寄存器分配。重叠的寄存器分配可能导致不可预测的 I/O 动作。

如果想通过获得错误检查来防止这种无效的寄存器分配的发生， 使用 ELadder 寄存器分配对话框建立寄存器分配。然后，将寄存器分配保存到梯形图文件（例如：文件名.lad），然后，用 C 程序将它下载，或者使用 **clearRegAssignment 和 addRegAssignment** 函数将寄存器分配发送给 C 程序。

在调用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**clearRegAssignment**

**例子**

```
#include “primitiv.h”
#include “rassign.h”

void main(void)
{
   request_resource(IO_SYSTEM);

   /*创建寄存器分配*/
   clearRegAssignment();

   addRegAssignment(AOUT_E105,1,40003,0,0,0);
   addRegAssignment(DIAG_forceLED,-1,100017,0,0,0);
   addRegAssignment(DIAG_controllerStatus,-1,30009,0,0,0);
   addRegAssignment(DIAG_protocolStatus,2,30010,0,0,0);
   release_resource(IO_SYSTEM);
}
```

# alarmIn

## 由消逝时间决定报警时间

**语法**

#include “rtc.h”

ALARM_SETTING alarmIn(unsigned hours, unsigned minutes, unsigned seconds)

**说明**

**alarmIn** 函数计算报警设置，根据从现在开始的 hours、minutes、seconds 计算出实时报警时间。

这个函数返回一个 ALARM_SETTING 结构，能够直接供 **setClockAlarm** 函数使用。这个结构指定了一个绝对报警，这个报警发生在由调用 **alarmIn** 函数而指定时间偏移。请参考在这一章开始处的 rtc.h 部分对 ALARM_SETTING 结构的域的描述。

**注**

如果 seconds 大于 60 秒，剩余时间将计入分钟。如果 minutes 大于 60 分钟，剩余时间将计入小时。

如果时间偏移超过一天，那么报警时间将越过当前日期。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**getClockAlarm,setClockAlarm,resetClockAlarm**

**例子**

```
#include “rtc.h”
#include “system.h”
#include “primitiv.h”

/* ------------------------------------------------------
    conservePower
    conservePower 函数将控制器设置为睡眠 10 分钟
  ------------------------------------------------------ */

void conservePower(void)
{
  ALARM_SETTING alarm;
  /*  报警 10 分钟  */
  alarm = alarmIn(0, 10, 0);
  setClockAlarm(alarm);

  /*  将控制器设置为节电模式  */
  sleep();

  release_resource(IO_SYSTEM);
}
```

# allocate_envelope

## 从 RTOS 获得一个邮箱

**语法**

#include “primitiv.h”
envelope *allocate_envelope(void);

**说明**

**allocate_envelope** 函数从操作系统中获得一个邮箱。如果没有邮箱可用，这个任务将被锁住，直到有可用的邮箱。

allocate_envelope 函数返回一个指向邮箱的指针。

**注**

邮箱被用来在两个任务之间发送消息。RTOS 从一个自由邮箱池中分配邮箱。邮箱被释放后，返回到自由邮箱池。

应用程序必须确保不再使用的邮箱一定要释放。这样，邮箱才可以继续使用。

**参考**

**deallocate_envelope**

**例子**

```
#include “primitiv.h”
extern unsigned other_task_id;
void task1(void)
{
   envelope *letter;

   /* 向另一个任务发送一个消息 */
   /* 假定已经释放了这个邮箱 */

   letter = allocate_envelope();
   letter->destination = other_task_id;
   letter->type = DATA;
   letter->data=5;
   send_message(letter);

   /* 接收来自另一个任务的消息 */

   letter = receive_message();
   /*  …..在这里处理数据*/
   deallocate_envelope(letter);

   /*  …..任务的剩余部分 */
}
```

# auto_pid

## 自动执行一个 PID 块

**语法**

#include “pid.h”

unsigned auto_pid(unsigned block, unsigned period);

**说明**

**auto_pid** 例程配置一个以一个特定周期自动执行的 PID 控制块，period 以 0.1 秒为时间刻度。block 的范围必须是从 0 到 PID_BLOCKS-1。

period 设置为 0 将自动停止执行控制块。

**注**

请参考《ELadder PID 控制参考手册》中关于 PID 控制的详细说明。

控制块在运行前必须被正确的配置，否则，会产生不可预测的结果。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**set_pid,get_pid,clear_pid**

# check_error

## 获得当前任务的错误码

**语法**

#include “primitiv.h”

int check_error(void)

**说明**

**check_error** 函数返回当前任务的错误码。当错误发生时，错误码被各种各样的 I/O 例程设定。每个任务维护一个独立的错误码。

**注**

在标准 C 库中的一些例程通过全局变量 **errno** 返回错误。这个变量不是唯一的对应一个任务，在读取之前，有可能被其它任务修改。

**参考**

**report_error**

# checksum

## 计算 CRC 校验

**语法**

#include “csum.h”

unsigned checksum(unsigned char *start, unsigned char *end, unsigned algorithm);

**说明**

**checksum** 函数在内存中计算一个校验和。由 start 指向内存中的开始字节的位置。end 指向结束字节的位置。算法可以是下列之一：

| | |
|---|---|
| **ADDITIVE** | 16 位字节方向求和 |
| **CRC_16** | CRC-16 类型多项式校验和 |
| **CRC_CCITT** | CCITT 类型多项式校验和 |
| **BYTE_EOR** | 8 位字节方向异或 |

CRC 校验和使用 **crc_reverse** 函数。

**参考**

**crc_reverse**

**例子**

这个例子列出两种类型的校验和

```
#include “csum.h”
void checksumExample(void)
{
   char str[] = "This is a test";
   unsigned sum;

   /*  显示加法校验和  */
   sum = checksum(str,str+strlen(str), ADDITIVE);
   printf("Additive checksum: %u\r\n", sum);

   /*  显示 CRC_16 校验和  */
   sum = checksum(str,str+strlen(str), CRC_16);
   printf"CRC_16 checksum: %u\r\n", sum);
}
```

# checkSFTranslationTable

## 测试存储和传送配置错误

**语法**

```
#include “protocol.h”
struct SFTranslationStatus checkSFTranslationTable(void);
```

**说明**

**checkSFTranslationTable** 函数检查在地址翻译表中的全部入口的有效性。它检测下列错误：

这个函数返回一个 SFTranslationStatus 结构。关于 SFTranslationStatus 结构中域的说明，请参考这一章开始部分的 protocol.h 部分。在结果中的 code 域被设置为下列值。如果有错误，index 域设定为指向无效翻译地址。

| 结果代码 | 意义 |
|---|---|
| SF_VALID | 翻译全部无效 |
| SF_NO_TRANSLATION | 入口定义重发一个消息到同一个 |
| SF_PORT_OUT_OF_RANGE | 一个或多个串口索引无效 |
| SF_STATION_OUT_OF_RANGE | 一个或多个站无效 |

**参考**

**getSFTranslation,setSFTranslation,checkSFTranslationTable**

**例子**

请参考 setSFTranslation 函数的例子。

# clearAllForcing

## 清除全部强制

**语法**

#include “rassign.h”

void clearAllForcing(void);

**说明**

**clearAllForcing** 函数清除 I/O 数据库寄存器中的全部强制条件。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**setForceFlag,getForceFlag,overrideDbase**

# clear_errors

## 清除串口错误计数器

**语法**

#include “serial.h”

void clear_errors(FILE *stream);

**说明**

**clear_errors** 函数清除由 stream 指定的串口中的串口错误计数器。如果 stream 没有指向一个有效的串口，这个函数不起作用。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**get_status**

# clear_pid

## 清除 PID 块变量

**语法**

#include “pid.h”

void clear_pid(unsigned block);

**说明**

**clear_pid** 例程将指定的控制块中的全部变量设为 0。clear_pid 通常是控制块配置的第一步。block 必须在 **0** 到 **PID_BLOCKS**-1 之间。

**注**

请参考《**ELadder PID 控制参考手册**》中关于 PID 控制的详细说明。

当程序运行时，保存在 PID 块中的值没有被初始化；在电源错误和程序加载的过程中，一直保留它们的值。PID 块变量必须由用户程序初始化。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**set_pid,get_pid,auto_pid**

# clear_protocol_status

## 清除协议计数器

**语法**

#include “protocol.h”

void clear_protocol_status(FILE *stream);

**说明**

**clear_protocol_status** 函数函数清除由 stream 指定的串口中的串口错误和消息计数器。如果 stream 没有指向一个有效的串口，这个函数不起作用。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**get_protocol_status**

# clearRegAssignment

## 清除寄存器分配

**语法**

#include “rassign.h”

void clearRegAssignment(void);

**说明**

**clearRegAssignment** 函数清除当前寄存器分配。在使用 **addRegAssignment** 函数生成新寄存器分配之前，请先调用这个函数。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**addRegAssignment**

**例子**

请参考 addRegAssignment 中的例子

# clearSFTranslationTable

## 清除存储和传送翻译配置

**语法**

#include “protocol.h”
void clearSFTranslationTable(void);

**说明**

**clearSFTranslationTable** 函数清除存储和传送翻译表中的全部入口。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**getSFTranslation,setSFTranslation,checkSFTranslationTable**

**例子**

请参考 setSFTranslation 中的例子

# clearStatusBit

## 清除控制器状态码的位

**语法**

#include “iohw.h”
void clearStatusBit(unsigned bitMask);

**说明**

**clearStatusBit** 函数清除由 bitMask 指定的控制器状态码的位。当状态码非 0，STAT LED 按照码的位顺序闪烁。如果状态码是 0，STAT LED 关闭。

这个函数返回状态寄存器的值。

**注**

如果码不是 0，状态输出打开。更多的信息请参考《**硬件手册**》。

码的位顺序产生错误指示 LED 短和长的闪烁。1/10 秒的短闪烁表示位值为 0。大约 1/2 秒的长闪烁表示位值为 1。最低位最先输出。显示尽量少的位数，也就是说，所有高位的 0 将被忽略。在重复之前有 2 秒的停顿。

STAT LED 位于 E001 控制器板的左上角的位置。

状态码的位 0 和 1 被用于寄存器分配。

**参考**

**setStatusBit,setStatus,getStatusBit**

# clear_tx

## 清除串口发送缓冲区

**语法**

#include “serial.h”

void clear_tx(FILE *stream);

**说明**

**clear_tx** 函数清除由 stream 指定的串口中的串口发送缓冲区。如果 stream 没有指向一个有效的串口，这个函数不起作用。

**参考**

**get_status**

# crc_reverse

## 计算一个 CRC 校验和

**语法**

#include “csum.h”

unsigned crc_reverse(unsigned char *start, unsigned char *end, unsigned poly, unsigned initial);

**说明**

**crc_reverse** 函数使用反向算法计算内存中的一个 CRC 类型校验和。由 start 指向内存中的开始字节的位置。end 指向结束字节的位置。多项式发生器由 poly 指定。poly 可以是任何值，但是，应该仔细选择，以保证好的错误检测结果。校验和累加器在计算之前由参数 initial 初始化。

**注**

反向算法根据位的移动方向命名。位向最低位移动。这种算法在使用相同多项式的情况下，生成与经典的、正向算法不同的校验和。

**参考**

**checksum**

# create_task

## 生成一个新任务

**语法**

#include “primitiv.h”

int create_task(void *function, unsigned priority, unsigned type, unsigned stack);

**说明**

**create_task** 函数为一个任务分配堆栈空间，并任务放置在等待队列中。function 指定要执行的例程的开始地址。如果优先级大于当前的任务，任务将被执行。

priority 是生成的任务的执行优先级，范围是 1 到 4。4 级优先级帮助任务的运行调度。

type 指定当程序停止时，任务是否终止。type 的有效值是：

**SYSTEM** 当程序停止，系统任务不终止。

**APPLICATION** 当程序停止，应用任务终止。

这里推荐只生成 **APPLICATION** 类型的任务。

参数 stack 指定为任务分配多少堆栈块。每个堆栈块有 256 字节。

**create_task** 函数返回生成的任务的任务 ID（TID）。如果发生错误，返回-1。

**注**

关于任务的更多信息，请参考实时操作系统章节。

注意主任务、梯形图和 I/O 扫描任务的优先级是 1。如果产生的任务是连续运行过程码，产生的任务的优先级是 1，同时定期调用 release_processor；否则，其余的优先级是 1 的任务将被锁住。

有些任务，例如协议句柄，通过使用 wait_event 或 receive_message 函数来等待一个事件，这样的任务选择的优先级大于 1，不会阻塞其它低优先级的任务。

需要的堆栈块的数量依赖于任务所调用的函数，以及生成的局部变量的大小。大多数任务需要 2 个堆栈块。如果使用了 printf 函数，至少需要 4 个堆栈块。如果生成大结构和数组，针对这些限制，应该使用局部变量用法。在任务的源文件中，作为静态局部变量，大结构和数组通常会被很好的处理。（对于任务中所有的函数，变量是全局的，但是，不能被其它文件中的函数看到。）

剩余堆栈空间可以通过禁用未使用的协议来变成可用。关于更多的信息，请参考 Start_Up 函数结构或 set_protocol()函数。

**参考**

**end_task**

**例子**

```
#include “primitiv.h”
#include “database.h”
#include “mriext.h”
#include “iohw.h”
#include “system.h”

#define   TIME_TO_PRINT      20

void task1(void)
{
  int a,b;

  while(TRUE)
  {
```

```
        /* 任务 1 的循环体 --处理 I/O */
        request_resource(IO_SYSTEM);
        a = dbase(MODBUS,30001);
        b = dbase(MODBUS,30002);
        setdbase(MODBUS,40020,a * b);
        release_resource(IO_SYSTEM);

        /* 允许执行其它任务 */
        release_processor();
    }
}

void task2(void)
{
    while(TRUE)
    {
        /* 任务 2 的循环体 --事件句柄 */
        wait_event(TIME_TO_PRINT);
        printf("It's time for a coffee break\r\n");
    }
}

/* ------------------------------
    当应用停止时，shutdown 函数停止发送 TIME_TO_PRINT 事件的信号
    ------------------------------ */
void shutdown(void)
{
    endTimedEvent(TIME_TO_PRINT);
}


void main(void)
{
    TASKINFO taskStatus;

    /* 继续处理优先级为 1 的任务 */
    create_task(task1, 1, APPLICATION, 2);

    /* 事件句柄需要为 printf 函数分配大的堆栈 */
    create_task(task2, 3, APPLICATION, 4);

    /* 当这个任务结束时，建立任务退出句柄，停止事件信号的发送 */
    taskStatus = getTaskInfo(0);
    installExitHandler(taskStatus.taskID,shutdown);
```

```
   /* 每 10 秒钟发生一次开始时间事件 */
   startTimedEvent(TIME_TO_PRINT, 100);

   interval(0,10);
   while(TRUE)
   {
      /* 主任务循环体 */

      /* 其它进程代码 */

      /* 允许执行其它任务 */

      release_processor();
   }
}
```

# dbase

## 读取 I/O 数据库的值

**语法**

#include “database.h”

int dbase(unsigned type, unsigned address);

**说明**

**dbase** 函数从读取 I/O 数据库中读取一个值。type 指定数据库中寻址的方法。address 指定在数据库中的位置。下表列出有效的地址类型和范围：

| 类型 | 地址范围 | 寄存器大小 |
|---|---|---|
| MODBUS | 00001 到 NUMCOIL<br>10001 到 10000+NUMSTATUS<br>30001 到 30000+NUMINPUT<br>40001 到 40000+NUMHOLDING | 1 位<br>1 位<br>16 位<br>16 位 |
| LINEAR | 0 到 NUMLINEAR-1 | 16 位 |

关于更多信息，请参考 I/O 数据库和寄存器分配章节。

如果指定的寄存器当前被强制，**dbase** 返回寄存器的强制值。

当控制器重启时，I/O 数据库不被修改。它是一个永久存储区，在临时断电期间，一直保持。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**setdbase**

**例子**

```
#include “database.h”
#include “primitiv.h”

void main(void)
{
   int a;
   request_resource(IO_SYSTEM);

   /* 读取 MODBUS 状态输入点 */
   a = dbase(MODBUS,10001);

   /* 读取 16 位寄存器 */
   a = dbase(LINEAR,3020);

   /* 读取从第一个状态寄存器开始的 16 位寄存器 */
   a = dbase(LINEAR,START_STATUS);

   /* 读取第 5 个输入寄存器 */
   a = dbase(LINEAR,START_INPUT + 5);

   release_resource(IO_SYSTEM);
}
```

# deallocate_envelope

## 释放一个邮箱到 RTOS

**语法**

```
#include “primitiv.h”
void deallocate_envelope(envelope *penv);
```

**说明**

**deallocate_envelope** 函数释放一个由 penv 指向的邮箱，并将它放回到由操作系统维护的自由邮箱池。

**参考**

**allocate_envelope**

**例子**

请参考 allocate_envelope 函数的例子

# end_application

## 结束所有应用任务

**语法**

```
#include “primitiv.h”
```

void end_application(void)

**说明**

end_application 函数结束所有由 create_task 函数生成的 APPLICATION 类型的任务。任务使用的堆栈空间和资源被释放。

**注**

这个函数通常由通信协议使用，用来停止正在执行的应用程序，以便在内存中加载一个新的应用。

**参考**

**create_task,end_task**

# end_task

## 终止一个任务

**语法**

#include “primitiv.h”
void end_task(unsigned task_ID)

**说明**

end_task 函数终止由 task_ID 指定的任务。任务使用的堆栈空间和资源被释放。end_task 函数终止 **APPLICATION** 和 **SYSTEM** 类型的任务。

**参考**

**create_task,end_application,getTaskInfo**

# endTimedEvent

## 终止一个定时事件

**语法**

#include “system.h”
unsigned endTimedEvent(unsigned Event)

**说明**

**endTimedEvent** 函数停止触发一个由 startTimedEvent 函数初始化的定时事件。

如果事件信号被取消，这个函数返回 TRUE。

如果事件号无效，或者这个事件不是先前 startTimedEvent 函数初始化的定时事件，这个函数返回 FALSE。在这些情况下，这个函数没有任何效果。

**注**

有效事件的数值范围是 0 到 RTOS_EVENTS-1。在 primitiv.h 中定义的任何事件在一个应用程序中都不是有效的事件。

**参考**

**startTimedEvent,poll_event,wait_event,signal_event**

**例子**

请参考 startTimedEvent 函数的例子

# forceLed

## 设置强制 LED 的状态

**语法**

#include “iohw.h”
void forceLed(unsigned state)

**说明**

**forceLed** 函数设置 FORCE LED 的状态。state 可以是 LED_ON 或 LED_OFF。

**注**

FORCE LED 用来指示强制的 I/O。在应用程序中小心使用这个函数。

**参考**

**setStatus**

# getBootType

## 获得控制器启动状态

**语法**

#include “system.h”
unsigned getBootType(void);

**说明**

**getBootType** 函数返回控制器启动状态。可能的返回值是：

| | |
|---|---|
| **SERVICE** | 控制器以 SERVICE 模式启动 |
| **RUN** | 控制器以 RUN 模式启动 |

**例子**

```
#include “serial.h”
#include “protocol.h”
#include “primitiv.h”

void main(void)
{
   struct prot_settings settings;

   /* 禁用串口 1 的协议 */
   settings.type = NO_PROTOCOL;
   settings.station = 1;
   settings.priorit = 3;
   settings.SFMessaging = FALSE;
```

```
    request_resource(IO_SYSTEM);
    set_protocol(com1,&settings);
    release_resource(IO_SYSTEM);

    /*  显示启动状态信息  */
    printf("Boot type: %d\r\n", getBootType());
}
```

# getclock

## 读取实时时钟

**语法**

```
#include “rtc.h”
struct clock getclock(void);
```

**说明**

**getclock** 函数从实时时钟硬件中读取时间和日期。

**getclock** 函数返回一个包含时间和日期信息的 clock 结构。

**注**

**getclock** 函数返回的时间格式与由 EMC 支持的标准 UNIX 形式函数不兼容。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**setclock**

**例子**

```
#include “stdio.h”
#include “rtc.h”
#include “primitiv.h”
main(void)
{
    struct clock now;
    request_resource(IO_SYSTEM);
    now = getclock();                    /**/
    release_resource(IO_SYSTEM);
    printf("%2d/%2d/%2d",now.day,now.month,now.year);
    printf("%2d:%2d\r\n",now.hour,now.minute);
}
```

# getClockAlarm

## 读取实时时钟报警设定

**语法**

```
#include “rtc.h”
ALARM_SETTING getClockAlarm(void);
```

**说明**

**getClockAlarm** 函数返回在实时时钟里的报警设定。这个报警用于把控制器从睡眠模式中唤醒。

**注**

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

**alarmIn,setClockAlarm**

# getForceFlag

## 获得一个寄存器的强制标志状态

**语法**

#include “database.h”

BOOLEAN getForceFlag(unsigned type, unsigned address, unsigned *value);

**说明**

**getForceFlag** 函数把强制标志的值或者指定的数据库寄存器的位掩码拷贝到由 value 指针指向的整数。address 的有效范围由数据库寻址 type 决定。

强制标志值是 0 或者 1，或者一个 LINEAR 数字地址的 16 位掩码。

如果 address 或者寻址 type 无效，将返回 FALSE，同时，由 value 指针指向的整数为 0；否则，返回 TRUE。下表列出有效的地址类型和范围：

| 类型 | 地址范围 | 寄存器大小 |
|---|---|---|
| MODBUS | 00001 到 NUMCOIL | 1 位 |
| | 10001 到 10000+NUMSTATUS | 1 位 |
| | 30001 到 30000+NUMINPUT | 16 位 |
| | 40001 到 40000+NUMHOLDING | 16 位 |
| LINEAR | 0 到 NUMLINEAR-1 | 16 位 |

**注**

当控制器重启时，强制标志不被修改。强制标志位于一个永久存储区，在临时断电期间，一直保持。

关于更多信息，请参考 I/O 数据库和寄存器分配章节。

**参考**

**setForceFlag,clearAllForcing,overrideDbase**

**例子**

这个程序为寄存器获得强制标志，对于 16 位状态寄存器，位于线性地址 302（寄存器 10727 到 10752），对于保持寄存器，位于线性地址 1540（寄存器 40005）。

```
#include “database.h”
void main(void)
{
```

```
    unsigned flag,bitmask;

    getForceFlag(MODBUS, 40001, &flag);
    getForceFlag(LINEAR, 302, &bitmask);
    getForceFlag(LINEAR, 1540, &flag);
}
```

# getOutputsInStopMode

## 获得停止模式下的输出

**语法**

#include “rassign.h”

void getOutputsInStopMode(unsigned *doutsInStioMode, unsigned *aoutsInStopMode);

**说明**

**getOutputsInStopMode** 函数将输出控制标志拷贝到由 doutsInStopMode 和 aoutsInStopMode 指向的整数。

如果由 doutsInStioMode 指向的值是 TRUE，当梯形逻辑程序停止时，数字输出将保持在它们最后的状态。

如果由 doutsInStioMode 指向的值是 FALSE，当梯形逻辑程序停止时，数字输出将关闭。

如果由 aoutsInStopMode 指向的值是 TRUE，当梯形逻辑程序停止时，模拟输出将保持在它们最后的状态。

如果由 aoutsInStopMode 指向的值是 FALSE，当梯形逻辑程序停止时，模拟输出将为 0。

**参考**

**setOutputsInStopMode**

**例子**

请参考 setOutputsInStopMode 函数的例子。

# getPortCharacteristics

## 获得串口特性

**语法**

#include “serial.h”

unsigned getPortCharacteristics(FILE *stream, PORT_CHARACTERISTICS *pCharacteristics);

**说明**

**getPortCharacteristics** 函数获得由 stream 指向的串口所支持的特征的信息。如果 stream 没有指向一个有效的串口，这个函数不起作用并返回 FALS；否则，返回 TRUE。

**getPortCharacteristics** 函数将串口特性拷贝到一个由 pCharacteristics 指向的结构。

**注**

关于串口的详细信息，请参考串口通信章节。

关于 PORT_CHARACTERISTICS 结构的域的描述，请参考这一章开始部分的 serial.h 部分。

**参考**

**get_port**

**例子**

```
#include “serial.h”
void main(void)
{
   PORT_CHARACTERISTICS options;

   getPortCharacteristics(com3, &options);
   fprintf(com1, "Dataflow options: %d\r\n", options.dataflow);
   fprintf(com1, "Protocol options: %d\r\n", options.protocol);
}
```

# get_pid

## 获得一个 PID 变量

**语法**

```
#include “pid.h”
int get_pid(unsigned name, unsigned block);
```

**说明**

**get_pid** 函数一个 PID 控制块的值，name 必须是 pid.h 中的宏定义。block 的范围必须是从 0 到 **PID_BLOCKS**-1。

**注**

请参考《**ELadder PID 控制参考手册**》中关于 PID 控制的详细描述。

当程序运行时，保存在 PID 块中的值没有被初始化；在电源错误和程序加载的过程中，一直保留它们的值。PID 块变量必须由用户程序初始化。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**参考**

set_pid, auto_pid, clear_pid

# get_port

## 获得串口配置

**语法**

```
#include “serial.h”
struct pconfig *get_port(FILE *stream, struct pconfig *settings);
```

**说明**

**get_port** 函数获得 stream 的串口配置。如果 stream 没有指向一个有效的串口，这个函数不起作用。

**get_port** 函数将串口设置拷贝到一个由 settings 指向的结构，并返回指向这个结构的指针。

**注**

关于串口的详细信息，请参考串口通信章节。

关于 pconfig 结构的域的描述，请参考这一章开始部分的 serial.h 部分。

**参考**

**set_port**

**例子**

```
#include “serial.h”

void main(void)
{
   struct pconfig settings;

   getPort(com1, &settings);
   fprintf(com1, "Baud rate: %d\r\n", settings.baud);
   fprintf(com1, "Duplex:        %d\r\n", settings.duplex);
}
```

# getProgramStatus

## 获得程序状态标志

**语法**

```
#include “system.h”
unsigned getPorgramStatus(void);
```

**说明**

**getProgramStatus** 函数返回应用程序状态标志。当 C 程序通过程序装载器删除或下载到控制器，状态标志被设置为 **NEW_PROGRAM**。

应用程序可以通过 **setProgramStatus** 函数修改状态标志。

**参考**

**setProgramStatus**

**例子**

当这个程序第一次运行时，它将缺省报警限制保存到 I/O 数据库。在后续的执行中，使用数据库中的限制。在运行中，数据库中的限制可以通过通信协议修改。

```
#include “mriext.h”
#include “system.h”
#include “primitiv.h”
```

```
#define HI_ALARM              41000
#define ALARM_OUTPUT              1026

void main(void)
{
   int inputValue;

   if    (getProgramStatus() == NEW_PROGRAM)
   {
      /*  设置缺省报警限制*/
      request_resource(IO_SYSTEM);
      setdbase(MODBUS, HI_ALARM, 4000);
      release_resource(IO_SYSTEM);

      /*  从现在开始，使用数据库中的值  */
      setProgramStatus(PROGRAM_EXECUTED);
   }
   while (TRUE)
   {
      request_resource(IO_SYSTEM);

      /*  测试超出限制的输入  */
      if(ain(INPUT) > dbase(MODBUS, HI_ALARM)
         setdbase(MODBUS, ALARM_OUTPUT, 1);
      else
         setdbase(MODBUS, ALARM_OUTPUT, 0);

      release_resource(IO_SYSTEM);

      /*  允许其它任务运行  */
      release_processor();

   }
}
```

# get_protocol

## 获得协议配置

**语法**

```
#include “protocol.h”
struct prot_settings *get_protocol(FILE *stream, struct prot_settings *settings);
```

**说明**

**get_protocol** 函数获得 stream 的通信协议配置。如果 stream 没有指向一个有效的串口，这个函数不起作用。settings 必须指向一个协议结构 prot_settings。

**get_protocol** 函数将串口设置拷贝到一个由 settings 指向的结构，并返回指向这个结构的指针。

关于 prot_settings 结构的域的描述，请参考这一章开始部分的 serial.h 部分。

关于通信协议的详细信息，请参考串口通信章节。

**参考**

**set_protocol**

**例子**

这个程序显示 **com1** 的协议配置。

```
#include “protocol.h”
void main(void)
{
  struct prot_settings settings;

  get_protocol(com1,&settings);
  printf("Type:         %d\r\n", settings.type);
  printf("Station:    %d\r\n", settings.Station);
  printf("Priority: %d\r\n", settings.Priority);
}
```

# get_protocol_status

## 获得协议信息

**语法**

```
#include “protocol.h”
struct prot_status get_protocol_status(FILE *stream);
```

**说明**

**get_protocol_status** 函数返回 stream 的通信协议错误和消息计数器。如果 stream 没有指向一个有效的串口，这个函数不起作用。

关于通信协议的详细信息，请参考串口通信章节。

**参考**

**clear_protocol_status**

**例子**

这个程序显示 **com2** 的校验和的错误计数器。

```
#include “protocol.h”
void main(void)
{
  struct prot_status status;
```

```
    status = get_protocol_status(com2);
    printf("Checksum: %d\r\n", status.checksum_errors);
}
```

# getSFMapping

## 读取翻译表映射控制

**语法**

#include “protocol.h”

unsigned getSFMapping(void);

**说明**

**getSFMapping** 和 **setSFMapping** 函数已经不再有任何功能，但是为了向后兼容，仍然被保留。通过在寄存器分配中包含 CNFG_StoreAndForward 模块来在 I/O 数据库中指定一个存储和传送表。

**参考**

**addRegAssignment**

# getSFTranslation

## 读取存储和传送翻译表

**语法**

#include “protocol.h”

struct SFTranslation getSFTranslation(unsigned index);

**说明**

**getSFTranslation** 函数返回存储和传送地址翻译表的索引的入口。如果索引无效，返回一个禁用的表入口。

这个函数返回一个 SFTranslation 结构。在这一章开始部分的 protocol.h 部分有关于这个结构的买偶数。。

**参考**

**setSFTranslation, clearSFTranslationTable, checkSFTranslationTable**

**例子**

请参考 setSFTranslation 函数的例子。

# get_status

## 获得串口状态

**语法**

#include “serial.h”

struct pstatus *get_status(FILE *stream, struct pstatus *status);

**说明**

**get_status** 函数获得 stream 的串口错误计数器、I/O 线状态和 I/O 驱动器缓冲区信息。如果 stream 没有指向一个有效的串口，这个函数不起作用。status 必须指向一个串口状态结构 pstatus。

**get_status** 函数将串口状态拷贝到一个由 status 指向的结构，并返回指向这个结构的指针。

关于 pstatus 结构的域的描述，请参考这一章开始部分的 serial.h 部分。

关于通信协议的详细信息，请参考串口通信章节。

**参考**

**clear_errors**

**例子**

这个程序显示 **com1** 的桢和校验错误。

```
#include “serial.h”
void main(void)
{
   struct pstatus status;

   status = get_status(com1, &status);
   printf("Framing: %d\r\n", status.framing);
   printf("Parity:    %d\r\n", status.parity);
}
```

# getStatusBit

## 读取控制器状态码的位

**语法**

```
#include “iohw.h”
unsigned getStatusBit(unsigned bitMask);
```

**说明**

**getStatusBit** 函数返回由 bitMask 指定的控制器状态码的值。

**参考**

**setStatusBit, setStatus, clearStatusBit**

# getTaskInfo

## 获得一个任务的信息

**语法**

```
#include “primitiv.h”
TASKINFO getTaskInfo(unsigned taskID);
```

**说明**

**getTaskInfo** 函数返回由 taskID 指定的任务的信息。如果 taskID 为 0，函数则返回当前任务的信息。

**注**

如果指定的 taskID 不是一个有效的事件，返回的数据的所有域都为 0。调用函数应该检查 TASKINFO 结构的 taskID 域：如果它是 0，则其余的信息都无效。

关于 TASKINFO 结构的域的描述，请参考这一章开始部分的 primitiv.h 部分。

**例子**

这个程序显示全部有效任务的信息。

```
#include “string.h”
#include “mriext.h”
#include “serial.h”
#include “protocol.h”
#include “primitiv.h”

void main(void)
{
   struct prot_settings settings;
   TASKINFO taskStatus;
   unsigned task;
   char state[6][20];
   char type[2][20];

   /* 设置状态字符串 */
   strcpy(state[TS_READY], "Ready");
   strcpy(state[TS_EXECUTING], "Executing");
   strcpy(state[TS_WAIT_ENVELOPE], "Waiting for Envelope");
   strcpy(state[TS_WAIT_EVENT], "Waiting for Event");
   strcpy(state[TS_WAIT_MESSAGE], "Waiting for Message");
   strcpy(state[TS_WAIT_RESOURCE], "Waiting for Resource");

   /* 设置类型字符串 */
   strcpy(type[APPLICATION], "Application");
   strcpy(type[SYSTEM], "System");

   /* 禁用串口 1 的协议 */
   settings.type = NO_PROTOCOL;
   settings.station = 1;
   settings..priority = 3;
   settings.SFMessaging = FALSE;
   request_resource(IO_SYSTEM);
   set_protocol(com1, &settings);
```

```
  release_resource(IO_SYSTEM);

  /*显示全部任务的信息*/
  for (task = 0; task <= RTOS_TASKS; task++)
  {
    taskStatus = getTaskInfo(task);
    if (taskStatus.taskID != 0)
    {
      /* 显示有效任务的信息*/
      fprintf(com1, "\r\n\r\nInformation about task %d:\r\n",task);
      fprintf(com1, "      Task ID:     %d:\r\n",taskStatus.taskID);
      fprintf(com1, "      Priority:    %d:\r\n",taskStatus.priority);
      fprintf(com1, "      Status:      %d:\r\n",taskStatus.status);
      if (taskStatus.status == TS_WAIT_EVENT)
      {
        fprintf(com1, "      Event:       %d:\r\n",taskStatus.requirement);
      }
      if (taskStatus.status == TS_WAIT_RESOURCE)
      {
        fprintf(com1, "      Resource: %d:\r\n",taskStatus.requirement);
      }
      fprintf(com1, "      Error:       %d:\r\n",taskStatus.error);
      fprintf(com1, "      Type:        %d:\r\n",type[taskStatus.type]);
    }
  }

  while (TRUE)
  {
    /* 允许其它任务运行 */
    release_processor();
  }
}
```

# getWakeSource

## 获得从睡眠模式中唤醒的条件

**语法**

#include “system.h”

unsigned getWakeSource(void);

**说明**

**getWakeSource** 函数返回有效的唤醒资源的位掩码。有效的唤醒资源如下：

WS_REAL_TIME_CLOCK

WS_INTERRUPT_INPUT

WS_LED_POWER_SWITCH

WS_COUNTER_0_OVERFLOW
WS_COUNTER_1_OVERFLOW
WS_COUNTER_2_OVERFLOW

**参考**

**setWakeSource, sleep**

**例子**

下面的代码片段显示可用的唤醒资源。

```
unsigned enabled;
enabled = getWakeSource();
fputs("Enabled wake up sources:\r\n",com1);
if (enabled & WS_REAL_TIME_CLOCK)
    fputs("    Real Time Clock\r\n",com1);
if (enabled & WS_INTERRUPT_INPUT)
    fputs("    Interrupt Input\r\n",com1);
if (enabled & WS_LED_POWER_SWITCH)
    fputs("    LED Power Switch\r\n",com1);
if (enabled & WS_COUNTER_0_OVERFLOW)
    fputs("    Counter 0 Overflow\r\n",com1);
if (enabled & WS_COUNTER_1_OVERFLOW)
    fputs("    Counter 1 Overflow\r\n",com1);
if (enabled & WS_COUNTER_2_OVERFLOW)
    fputs("    Counter 2 Overflow\r\n",com1);
```

# install_handler

## 安装串口句柄

**语法**

```
#include “serial.h”
void install_handler(FILE *stream, void *function(unsigned, unsigned));
```

**说明**

**install_handler** 函数安装一个串口字符句柄函数。串口驱动器每接收一个字符就调用这个函数。如果 stream 没有指向一个有效的串口，这个函数不起作用。

function 指定一个有两个参数的句柄函数。第一个参数是一个接收的字符。第二个参数是一个错误标志。一个非 0 值表示有一个错误。如果 function 是 **NULL**，这个串口缺省的句柄将被安装。缺省的句柄什么也不做。

**注**

**install_handler** 函数可以被用来写定制的通信协议。

在完成接收器中断句柄之后，调用这个句柄。在中断句柄处理期间，RTOS 调用（见本章开始部分的实时操作系统函数的函数列表）不能使用，除了一个例外。

**interrupt_signal_event** 的 RTOS 函数调用可以用信号报告一个事件。

为了优化性能， 应当使 com3 和 com4 的消息长度最小。推荐 com3 和 com4 只用于操作员显示终端，以及程序调试和分析。

**例子**

```
#include “mriext.h”
#include “primitiv.h”
#include “serial.h”
#include “protocol.h”

#define CHAR_RECEIVED 11

/* ----------------------------------------
    signal

    当 com1 收到一个字符，这个例程触发一个事件。如果有错误，这个字符就被忽略。
    ---------------------------------------- */

void signal(unsigned character, unsigned error)
{
  if (error == 0)
      interrupt_signal_event( CHAR_RECEIVED );

    character;
}

/* ----------------------------------------
    main

    这个程序显示 com1 收到全部字符，使用一个安装的句柄发出字符接收信号
    ---------------------------------------- */

void main(void)
{
  struct prot_settings protocolSettings;
  int character;

  /* 禁用协议 */
  get_protocol(com1, &protocolSettings);
  protocolSettings.type = NO_PROTOCOL;
  request_resource(IO_SYSTEM);
  set_protocol(com1, &protocolSettings);
  release_resource(IO_SYSTEM);
```

```
   /* 字符句柄可用 */
   install_handler(com1,signal);

   /* 显示每一个接收的字符 */
   while(TRUE)
   {
        wait_event(CHAR_RECEIVED);
        character = fgetc(com1);
        fputs("character: ", com1);
        fputc(character, com1);
        fputs("\r\n", com1);
   }
}
```

# installClockHandler

## 安装实时时钟句柄

**语法**

#include “rtc.h”
void installClockHandler(void (*function) (void));

**说明**

**installClockHandler** 函数安装一个实时时钟报警句柄函数。每次实时时钟报警发生时，实时时钟报警句柄函数调用这个函数。

function 指定一个句柄函数。如果 function 是 **NULL**，这个句柄将被禁用。

**注**

在中断句柄处理期间，RTOS 调用（见本章开始部分的实时操作系统函数的函数列表）不能使用，除了一个例外。**interrupt_signal_event** 的 RTOS 函数调用可以用信号报告一个事件。

**参考**

**setClockAlarm, resetClockAlarm**

**例子**

```
/* ----------------------------------------
    这个程序演示怎样在一天的特定时间调用一个函数。
    ---------------------------------------- */

#include “mriext.h”
#include “serial.h”
#include “protocol.h”
#include “rtc.h”
```

```
#include “primitiv.h”

#define ALARM_EVENT 20

/* ----------------------------------------
    当报警产生时，这个函数触发一个事件。
    ---------------------------------------- */

void alarmHandler(void)
{
    interrupt_signal_event( ALARM_EVENT );
}

/* ----------------------------------------
    这个任务处理时钟句柄产生的报警信号
    ---------------------------------------- */
void processAlarms(void)
{
    while(TRUE)
    {
        wait_event(ALARM_EVENT);

        /* 重置下一天的报警 */
        request_resource(IO_SYSTEM);
        resetClockAlarm();
        release_resource(IO_SYSTEM);

        fprintf(com1, "It's quitting time!\r\n");
    }
}

void main(void)
{
   struct prot_settings settings;
   ALARM_SETTING alarm;

   /* 禁用 com1 的协议 */
   settings.type = NO_PROTOCOL;
   settings.station = 1;
   settings.priority = 3;
   settings.SFMessaging = FALSE;
   request_resource(IO_SYSTEM);
   set_protocol(com1, &Settings);
   release_resource(IO_SYSTEM);
```

```
  /* 安装时钟句柄函数 */
  installClockHandler(alarmHandler);

  /* 产生任务处理报警事件 */
  create_task(processAlarms, 3, APPLICATION, 4);

  /* 设置实时时钟报警 */
  alarm.type   = AT_ABSOLUTE;
  alarm.hour   = 16;
  alarm.minute = 0;
  alarm.second = 0;

  request_resource(IO_SYSTEM);
  setClockAlarm(alarm);
  release_resource(IO_SYSTEM);

  while(TRUE)
  {
    /* 主任务循环体 */

    /* 其它进程代码 */

    /* 允许其它任务运行 */
    release_processor();
  }
}
```

# installExitHandler

## 安装任务结束时被调用的句柄

**语法**

#include “primitiv.h”

unsigned installExitHandler(unsigned *taskID*, void (*function) (void);

**说明**

**installExitHandler** 函数定义一个任务结束时被调用的函数，这个任务由 taskID 指定。function 指定一个句柄函数。如果 function 是 **NULL**，这个句柄将被禁用。

**注**

退出句柄函数在下列情况下被调用：

由 end_task 函数结束的任务

end_application 函数被执行，同时，这个函数是是一个 APPLICATION 类型的函数

程序被 ELadder 程序停止，同时，这个函数是是一个 APPLICATION 类型的函数

C 程序被 ELadder 程序删除

**例子**

请参考 startTimedEvent 函数的例子

# interruptInput

## 读取中断数字输入的状态

**语法**

#include “iohw.h”
#include “mriext.h”
unsigned interruptInput(void);

**说明**

**interruptInput** 函数读取控制器的中断数字输入点的状态。如果输入被激活，就返回 **TRUE**，否则，返回 **FALSE**。

**注**

中断输入可以用做控制器的唤醒资源或者一个附加的数字输入。关于线路的细节，请参考《硬件手册》。

# interrupt_signal_event

## 在一个中断句柄中用信号通知一个事件

**语法**

#include “primitiv.h”
void interrupt_signal_event(unsigned *event_number*)

**说明**

**interrupt_signal_event** 函数用在一个中断句柄中用以表示事件的发生。这个函数触发表示 *event_number* 事件已经发生。

如果有多个任务等待事件，最高优先任务将被执行。否则，事件标志将增加。一个事件发生超过 255 次将被记录。如果有更高优先的任务等待事件，当前任务将被锁住。

**注**

关于事件的详细信息，请参考实时操作系统章节。

这个函数只能被用在一个中断句柄里。

有效的事件范围是 0 到 RTOS_EVENTS-1。任何在 primitiv.h 中定义被用于应用程序的事件都不是一个有效的事件。

**参考**

**signal_event,startTimeEvent,installClockHandler**

# interval

## 设置计时器间隔

**语法**

#include “iohw.h”

void interval(unsigned timer, unsigned value);

**说明**

**interval** 函数设置计时器间隔为 value。间隔为这个数值和 0.1 秒相乘。

如果计时器数字无效，任务错误代码被设为 **TIMER_BADTIMER**。

**注**

缺省的计时器间隔是 1/10 秒。

**参考**

**settimer, read_timer_info, check_error**

**例子**

设置计时器 5 计数 12 秒，每 1 秒滴答一次。

```
interval(5, 10);       /* 1.0 秒  滴答  */
settimer(5, 12);       /*  时间  = 12 秒  */
interval(5, 1);       /* 0.1 秒  滴答  */
settimer(5, 120);       /*  时间  = 12 秒  */
```

# jiffy

## 读取系统时钟

**语法**

#include “rtc.h”

unsigned long jiffy(void)

**说明**

**jiffy** 函数返回系统精确时钟的当前值。时钟每 1/60 秒增加一次。时钟到达 5183999 之后变成 0。这是一天的 1/60 秒的倍数。

**注**

实时时钟和精密时钟不相关。经过几天之后，他们可能有轻微的差异。

使用精细时钟测量比 1/10 秒的计时器更精密的时间。

**参考**

**timer,interval,setjiffy**

**例子**

这个程序用 jiffy 计时器决定一小段程序的运行时间。这个部分运行 10 次，来为测量提供一个长时间基数。

```
#include “stdio.h”
#include “rtc.h”
void main(void)
{
    int iterations =10;
    int i;

    setjiffy(0UL);
    for(i=0;i<=iterations; i++)
    {
      /* 时间声明 */
    }
    printf("average time=%ld jiffies", jiffy()/iterations);
}
```

# ledGetDefault

## 读取 LED 电源控制参数

**语法**

```
#include “iohw.h”
struct ledControl_tag ledGetDefault(void);
```

**说明**

**ledGetDefault** 函数返回缺省的 LED 电源控制参数。EchoI/O 模块的 LED 电源由控制器控制。为了省电，LED 可以被禁用。

用户可以通过控制器上的 LED POWER 开关来改变 LED 电源设置。经过一段用户指定的时间之后，LED 电源返回缺省的状态。

**例子**

请参考 ledSetDefault 函数的例子。

# ledPower

## 设置 LED 电源状态

**语法**

```
#include “iohw.h”
unsigned ledPower(unsigned state);
```

**说明**

**ledPower** 函数设置 LED 电源状态。LED 电源将一直保持这个状态，直到缺省的延迟时间结束。state 必须是 LED_ON 或 LED_OFF。

如果状态有效，这个函数返回 TRUE；否则，返回 FALSE。

**注**

LED 电源也可以由 LED POWER 开关来控制。用户可以通过开关取消这个函数的设置。

**ledSetDefault** 函数设置缺省的 LED 电源状态。这个状态可以取消 ledPower 函数的设置。

**参考**

**ledPowerSwitch,ledSetDefault,ledGetDefault**

# ledPowerSwitch

## 读取 LED 电源开关的状态

**语法**

#include “iohw.h”
#include “mriext.h”
unsigned ledPowerSwitch(void);

**说明**

**ledPowerSwitch** 函数返回 LED 电源开关的状态。如果开关已经释放，这个函数返回 FASLE；如果开关按下，这个函数返回 TRUE。

**注**

这个开关可以被程序使用，用于用户输入。但是，按下这个开关会有改变 LED 电源状态这样的副作用。

**参考**

**ledPower,ledSetDefault,ledGetDefault**

# ledSetDefault

## 设置 LED 电源控制的缺省参数。

**语法**

#include “iohw.h”
#include “mriext.h”
unsigned ledSetDefault(struct ledControl_tag ledControl);

**说明**

**ledSetDefault** 例程设置 LED 电源控制的缺省参数。EchoI/O 模块的 LED 电源由控制器控制。为了省电，LED 可以被禁用。

用户可以通过控制器上的 LED POWER 开关来改变 LED 电源设置。经过一段用户指定的时间之后，LED 电源返回缺省的状态。

ledControl 结构包含缺省的值。请参考在这一章开始处的 iohw.h 部分对 ledControl_tag 结构的域的描述。有效的 state 域的值是 LED_ON 和 LED_OFF。有效的 time 域的值是 1 到 65535 分钟。

如果参数有效，这个函数返回 TRUE；否则，这个函数返回 FALSE。如果两个参数都无效，缺省的值不改变。

在使用这个函数之前，必须申请 IO_SYSTEM 资源。

**例子**

```
#include “iohw.h”
#include “mriext.h”
#include “primitiv.h”
void main(void)
{
    struct ledControl_tag ledControl;

    request_resource(IO_SYSTEM);

    /* 20 分钟后，关闭 LED */
    ledControl.time    = 20;
    ledControl.state = LED_OFF;
    ledSetDefault(ledControl);

    release_resource(IO_SYSTEM);

    /*  程序的其余部分  */
}
```

# master_message

## 发送协议命令

**语法**

```
#include “protocol.h”
extern unsigned master_message(FILE *stream, unsigned function, unsigned slave, unsigned address, unsigned master_address, unsg length);
```

**说明**

**master_message** 函数发送通信协议命令。通信协议任务等待从站的响应。当前任务继续执行。

stream 指定串口。

function 指定协议功能码。

slave 指定从站的网络地址。这个就是指从站号。

address 指定从站的数据的地址。依照于协议功能码，可能是从这个地址读数据，也可能是写数据到这个地址。

master_address 指定主站的数据的地址（这个控制器）。依照于协议功能码，可能是从这个地址读数据，也可能是写数据到这个地址。

length 指定数字或者寄存器。

**master_message** 函数协议从协议驱动器返回命令的状态。

| 值 | 描述 |
|---|---|
| MM_SENT | 消息被发送到从站 |
| MM_BAD_FUNCTION | 无法识别函数 |
| MM_BAD_SLAVE | 从站地址无效 |
| MM_BAD_ADDRESS | 主数据库或从数据库地址无效 |
| MM_BAD_LENGTH | 指定太多或太少的寄存器 |

**注**

更多信息请参考通讯控制手册。

调用此函数前必须调用资源 IO_SYSTEM

**参考**

get_protocol_status, clear_protocol_status

**例子**

下面程序使用 Modbus 协议，通过 com2 端口发送一个控制信息，然后等待回应。成功的和失败的信息数通过 com1 端口输出

```
/*-----------------------------------
poll.c
-----------------------------------*/
#include “mriext.h”
#include “serial.h”
#include “protocol.h”
#include “primitiv.h”
#include “iohw.h”
#include “.h”
/*-------------------------------------------------------
wait_for_response
函数 wait_for_response 等待联到串口并接收到控制信号
由 stream 指定的元件的响应。当接收到响应或响应周期
（由 time 指定）结束函数返回
-------------------------------------------------------*/

void wait_for_response(FILE *stream, unsigned time)
{
    struct prot_status status;
    static unsigned long good, bad;

    interval(0,1);
    settimer(0,time);
    do{
        /*允许其它任务执行*/
        release_processor();
        status=get_protocol_status(stream);
```

```
        }
        while(timer(0) && status.command == MM_SENT);

        if(status.command == MM_RECEIVED)
            good++;
        else
            bad++;
        fprintf(com1,"Good: %8lu Bad: %8lu\r", good, bad);
}
/*-----------------------------------------
        main
        main 函数设置串口并发送命令给从属 Modbus
-----------------------------------------*/
void main(void)
{
        struct prot_settings settings;
        struct pconfig portset;

        request_resource(IO_SYSTEM);

        /*使协议在串口 1 上无效*/
        settings.type=NO_PROTOCOL;
        settings.station=1;
        settings.priority=3;
        settings.SFMessaging=FALSE;
        set_protocol(com1, &settings);

        /*设置 com1 上的通讯参数*/
        portset.baud        =BAUD9600;
        portset.duplex      =FULL;
        portset.parity      =NONE;
        portset.data_bits   =DATA8;
        portset.stop_bits   =STOP1;
        portset.flow_rx         =DISABLE;
        portset.folw_tx         =DISABLE;
        portset.type        =RS232;
        portset.timeout         =600;
        set_port(com1, &portset);

        /*在 com2 上设置 Modbus 协议*/
        setting.type        =MODBUS_ASCII;
        setting.station     =2;
        setting.priority=3;
        setting.SFMessaging     =FALSE;
```

```
    set_protocol(com2, &settings);

    /*设置 com2 上的通讯参数*/
    portset.baud        =BAUD9600;
    portset.duplex      =HALF;
    portset.parity      =NONE;
    portset.data_bits   =DATA8;
    portset.stop_bits   =STOP1;
    portset.flow_rx         =DISABLE;
    portset.folw_tx         =DISABLE;
    portset.type        =RS485_2WIRE;
    portset.timeout         =600;
    set_port(com2, &portset);

    release_resource(IO_SYSTEM);

    /*主通讯循环*/
    while(TRUE)
    {
        /*转发从属输入到输出*/
        request_resource(IO_SYSTEM);
        master_message(com2,2,1,10001,17,8);
        release_resource(IO_SYSTEM);
        wait_for_response(com2, 10);

        /*转发从属输入到输出*/
        request_resource(IO_SYSTEM);
        master_message(com2,15,1,1,10009,8);
        release_resource(IO_SYSTEM);
        wait_for_response(com2, 10);

        /*允许其它任务执行*/
        release_processor();
    }
}
```

# modbusExceptionStatus

## 设置对协议命令的响应

**语法**

#include “protocol.h”

void modbusExceptionStatus(unsigned char *status*);

**说明**

函数 **modbusExceptionStatus** 用在连接 Modbus 兼容通讯协议上。在响应**读异常**命令时他被赋值。

*status* 的值由主机的需求决定。

**注**

每次协议命令被收到后，指定的结果将被发送，直到新的结果被指定。

当控制器复位时，结果被清除。每次应用程序运行时必须初始化此状态。

**参考**

**modbusSlavelD**

# modbusSlaveID

## 设置协议命令响应

**语法**

#include “protocol.h”

void modbusSlaveID(unsigned char * *string* , unsigned *length*);

**说明**

函数 **modbusSlaveID** 用在连接 Modbus 兼容通讯协议上。在响应**报告从属 ID** 命令时他被赋值。

*String* 表示长度至少为 *length* 的字符串，字符串的内容由主机决定。字符串不是以 NULL 结束，中间可以包含多个 NULL 字符。

*length* 指出协议命令返回多少个字符。*length* 的范围必须在 1---REPORT_SLAVE_ID_SIZE。

如果 *length* 太大，则只有头 REPORT_SLAVE_ID_SIZE 各字符被发送。

**注**

每次协议命令收到后，指定的结果将不断发送，直到新的结果被指定。

数据拷贝到 *string* 中。

当控制器复位时，结果被清除。每次应用程序运行时必须初始化从属 ID 字符串。

**参考**

**modbusExceptionStatus**

# modemDial

## 连接远程拨号控制器

**语法**

#include “dialup.h”

enum DialError modemDial(struct ModemSetup * *configuration*, reserve_id **id*)

**说明**

函数 **modemDial** 使用外接拨号 Modem 使一个控制器和一个远程控制器连接。

**modemDial** 函数可以每一个串口上执行。函数 **modemDial** 处理所有的串口共享和重复拨号操作。

通过 ModemSetup 结构的参数 *configuration* 定义拨号的串口、拨号参数、Modem 初始化字符串和电话号码。参阅本章 dialup.h 小节以了解 ModemSetup 结构。

参数 *id* 指向一个串口保留标识。串口标示符可以保证没有其它 Modem 访问此串口。必须以此参数提供给函数 **modemDialEnd** 和 **modemDialStatus** 中。

函数的返回值是一个错误码。DE_NoError 表示连接已经开始。其它情况都表示发生一个错误。请参阅 dialup.h 小节来查看关于错误码的完整说明。

**注**

串口类型必须设置为 RS232_MODEM

函数 **modemDialStatus** 返回由 **modemDial** 设置的连接操作的状态。

函数 **modemDialEnd** 结束同远程控制器的连接。在结束一个连接和重新连接之间必须暂停几秒钟。此暂停保证了外接 Modem 挂断。

如果在串口初始化后串口通讯协议已经启动，则此通讯协议会暂时禁止，直到完成后连接才重新使能。当用函数 **modemDialEnd** 中断连接后，协议设置将被重新复原。

如果一个串口正在使用 **modemInit** 函数或正有一个接入呼叫，函数 **modemDial** 不能访问端口并且返回一个错误码 DE_NotInControl。

为了优化性能， 应当使 com3 和 com4 的消息长度最小。推荐 com3 和 com4 只用于操作员显示终端，以及程序调试和分析。

**参考**

**modemDialEnd**, **modemDialStatus**, **modemInit**

**例子**

请参阅使用拨号 Modem 小节的例子

# modemDialEnd

## 中断拨号连接

**语法**

#include “dialup.h”

void modemDialEnd(FILE **port*, reserve_id **id*, DialError **error*);

**说明**

函数 **modemDialEnd** 中断拨号连接或正在进行中的连接。*port* 指定 Modem 连接的串口。

*id* 是由函数 modemDial 返回的保留串口标示符。

函数给变量 *error* 赋值。如果没有错误则返回 DE_NoError。其它值表示有错误发生。请参阅 dialup.h 小节以了解错误码的完整描述。

**注**

串口类型必须设置为 RS232_MODEM

一个连接可以被下面的事件中断。

6.1 执行 **modemDialEnd** 函数

6.2 远程设备挂断电话线

6.3 由于电话线的问题使得载波信号丢失

在结束一个连接和初始化连接之间必须暂停几秒钟。此暂停保证了外接 Modem 挂断。

标示符一直有效，直到呼叫中断或者另外的 modem 函数或接入呼叫控制着这个串口。

如果另外一 modem 函数或接入呼叫控制着这个串口，则函数 **modemDialEnd** 返回一错误码

DE_NotInControl。其它值表示有错误发生。请参阅 dialup.h 小节以了解错误码的完整说明。

**参考**

**modemDial**, **modemDialStatus**

# modemDialStatus

## 返回拨号连接状态

**语法**

#include “dialup.h”

void modemDialStatus(FILE *_port_, reserve_id _id_, enum DialError * _error_, enum DialState *_state_);

**说明**

函数 **modemDialStatus** 返回由函数 **modemDial** 初始化的远程连接的状态 。*port* 指示 Modem 连接的串口。*id* 是由函数 **modemDial** 返回的端口保留标识符。

函数为变量 *error* 设值。如果没有错误发生则返回 DE_NoError。请参阅 dialup.h 小节以了解错误码的完整描述。

函数为变量 *state* 设值。如果错误码为 DE_NotInControl，变量 state 的值无效.请参阅 dialup.h 小节以了解状态码的完整描述。

**注**

串口类型必须设置为 RS232_MODEM

标示符一直有效，直到呼叫中断或者另外的 modem 函数或接入呼叫控制着这个串口。

如果另外一 modem 函数或接入呼叫控制着这个串口，则函数 **modemDialEnd** 返回一错误码

DE_NotInControl。其它值表示有错误发生。请参阅 dialup.h 小节以了解错误码的完整说明。

**参考**

**modemDial**, **modemDialEnd**

# modemInit

## 初始化拨号 Modem

**语法**

#include “dialup.h”

enum DialError modemInit(struct ModemInit *_configuration_, reserve_id _id_);

**说明**

函数 **modemInit** 发送一个初始化字符串给外接 Modem。一般将 Modem 设置为应答接入呼叫。

**modemInit** 函数可以用在任一串口上。函数 **modemInit** 处理所有的串口共享和重复拨号。

通过 ModemInit 结构的参数 *configuration* 定义拨号的串口、拨号参数、Modem 初始化字符串和电话号码。参阅本章 dialup.h 小节以了解 ModemInit 结构。

参数 *id* 设为串口的保留标示符。此标示符确保没有其它 Modem 控制函数访问此串口。此参数必须提供给函数 **modemInitStatus** 和 **modemInitEnd** 中

则函数返回一错误码 DE_NotError 表示初始化已经开始。其它值表示有错误发生。请参阅 dialup.h 小节以了解错误码的完整描述。

**注**

串口类型必须设置为 RS232_MODEM。

函数 **modemInitStatus** 返回由函数 **modemInit** 设定的联接状态。

函数 **modemInitEnd** 中断 Modem 的初始化。

如果在 Modem 初始化时串口通讯协议已经启动，则此通讯协议会暂时禁止，直到完成初始化后才重新使能连接。当用函数 **modemInitEnd** 中断初始化后，协议设置将被重新复原。

如果函数 **modemDial** 或接入呼叫正在端口上进行，函数 **modemInit** 不能访问此端口并且返回一个错误码 DE_NotInControl。

**参考**

**modemInitEnd**, **modemInitStatus**, **modemDial**

**例子**

参考使用拨号 Modem 小节的例子。

# modemInitEnd

## 退出拨号 Modem 的初始化

**语法**

#include “dialup.h”

void modemInitEnd(FILE *_port_, reserve_id _id_, enum DiaError *_error_);

**说明**

函数 **modemInitEnd** 中断初始化。参数 *port* 指定 Modem 安装的串口。*id* 是由函数 **modemInit** 返回的端口的标示符。

函数给参数 *error* 赋值。如果没有错误则返回 DE_NoError。其它值表示有错误发生。请参阅 dialup.h 小节以了解错误码的完整描述。

**注**

串口类型必须设置为 RS232_MODEM。

正常情况下，一旦函数 **modemInitStatus** 检测到初始化完成后，函数被执行一次。

保留标示符直到初始化完成或中断，并且其它 Modem 函数或接入呼叫控制端口后才有效。如果其它 Modem 函数或接入呼叫控制着端口则函数 **modemInitEnd** 返回错误码 DE_NotInCoutrol。

**参考**

**modemInit**, **modemInitStatus**

# modemInitStatus

## 返回拨号 Modem 初始化状态

**语法**

#include “dialup.h”

void modemInitStatus(FILE ***port*, reserve_id *id*, enum DialError ***error*, enum DialState ***state*);

**说明**

函数 **modemInitStatus** 返回由函数 **modemInit** 启动的初始化的状态。参数 port 指定端口。*Id* 是由函数 **modemInit** 返回的端口保留标示符。

函数为变量 *error* 赋值。如果没有错误则返回 DE_NoError。其它值表示有错误发生。请参阅 dialup.h 小节以了解错误码的完整描述。

函数返回当前拨号状态到变量 *state*。如果错误码为 DE_NotInControl，则状态值无效。参阅 dialup.h 小节以了解状态码的完整描述。

**注**

串口类型必须设置为 RS232_MODEM。

端口将保持 DS_Calling 状态，直到 Modem 初始化完成或失败。在等待返回码不为 DS_Calling 后才能调用函数 **modemInitEnd**。

保留标示符直到初始化完成或中断，并且其它 Modem 函数或接入呼叫控制端口后才有效。

**参考**

**modemInit**, **modemInitEnd**

# overrideDbase

## 向强制 I/O 数据库中写值

**语法**

#include “database.h”

unsigned overrideDbase(unsigned *type*, unsigned *address*, int *value*);

**说明**

函数 **overrideDbase** 向 I/O 数据库中写值，即使数据库寄存器被强制也能被写入。参数 *type* 指定数据库寻址方式。*address* 指定数据库地址。

如果寄存器被强制，则寄存器保持强制，但是强制值更新。

如果地址或寻址方式无效，I/O 数据库不改变并且返回 FALSE; 其它情况返回 TRUE; 下表列出正确寻址方式和范围

| 类型 | 地址范围 | 寄存器大小 |
|---|---|---|
| MODBUS | 00001 到 NUMCOIL | 1 比特 |
| | 10001 到 10000+NUMSTATUS | 1 比特 |
| | 30001 到 30000+NUMINPUT | 16 比特 |
| | 40001 到 40000+NUMHOLDING | 16 比特 |
| LINEAR | 0 到 NUMLINEAR-1 | 16 比特 |

**注**

当以 LINEAR 方式写数据库数字 I/O 部分时，一次将会写入 16 位数据。

控制器复位时 I/O 数据库不变。这是一个永久存储区，掉电数据不变。

更多信息请参阅 I/O 数据库和寄存器分配章节。

调用函数前必须申请资源 IO_SYSTEM

**参考**

**setdbase**, **dbase**, **setForceFlag**

**例子**

```
#include “primitiv.h”
#include “database.h”
void main(void)
{
    request_resource(IO_SYSTEM);

    overrideDbase(MODBUS, 40001,102);
    overrideDbase(LINEAR, 302,330);

    release_resource(IO_SYSTEM);
}
```

# poll_event

## 检测事件的发生

**语法**

#include “primitiv.h”

int poll_event(int *evnet*);

**说明**

函数 poll_event 检测事件的发生。如果发生事件，函数 poll_event 返回值为 TRUE，并且事件计数器数量减一，其它情况返回 FALSE。

当前任务继续执行。

**注**

关于事件的更多信息请参阅实时操作系统章节。

有效的事件数范围为 0 到 RTOS_EVENTS-1，在应用程序中使用 primitiv.h 中定义的事件不能用于用户程序中。

**参考**

**signal_event, wait_event, startTimedEvent**

**例子**

```
#include “mriext.h”
#include “primitiv.h”
void main(void)
{
    while(TRUE)
    {
        if(poll_event(COM1_RCVR))
            fputc(getc(com1), com2);
        if(poll_event(COM2_RCVR))
            fputc(getc(com2), com1);
    /*允许其它任务执行*/
    release_processor();
    }
}
```

# poll_message

## 检测收到信息

**语法**

#include “primitiv.h”

envelope *poll_message(void);

**说明**

poll_message 检测当前任务是否收到信息。如果收到信息，函数 poll_message 返回一个指向 envelope 的指针。如果没有信息则返回 NULL。

当前任务继续执行。

**注**

关于信息的更多内容请参阅实时操作系统章节。

**参考**

send_message, receive_message

**例子**

这个例子连续执行并处理收到的信息

```
#include “mriext.h”
#include “primitiv.h”
void task(void)
{
    envelope *letter;
    while(TRUE)
    {
        letter=poll_message();
        if(letter!=NULL)
            /*处理信息*/
            /*在此加入代码*/
    }
}
```

# poll_resource

## 检测资源

**语法**

```
#include “primitiv.h”
int poll_resource(int resource);
```

**说明**

函数 **poll_resource** 检测由变量 *resource* 指定的资源是否可用。如果资源可用则它分配给一个任务。

如果资源可用，函数 **poll_resource** 返回 TRUE，否则返回 FALSE。

当前任务继续执行。

**注**

关于资源的更多信息请参阅实时运算系统的有关章节。

**参考**

**request_resource, release_resource**

# portConfiguration

## 获得端口配置结构指针

**语法**

#include “serial.h”

struct pconfig *portConfiguration(FILE **stream*);

**说明**

函数 **portConfiguration** 返回配置结构指针。如果参数 *stream* 无效则返回 NULL。

**注**

推荐使用 **get_port** 和 **set_port** 来访问配置表。

# portIndex

## 得到串口索引

**语法**

#include “serial.h”

unsigned portIndex(FILE **stream*);

**说明**

函数 **portIndex** 返回参数 *stream* 指定的串口阵列的索引。

如果 *stream* 参数无效，则返回 SERIAL_PORTS，表明有一个错误。

**参考**

**portStream**

# portStream

## 得到由索引指定的串口

**语法**

#include “serial.h”

FILE *portStream(unsigned *index*);

**说明**

函数 **portStream** 返回由参数 *index* 指定的文件指针。此函数和函数 **portIndex** 相对应。

如果参数 *index* 无效则返回 NULL。

**参考**

**portIndex**

# pulse

## 产生一个方波

**语法**

#include “iohw.h”
void pulse(unsigned *channel*, unsigned *bit*, unsigned *timer*, unsigned *on*, unsigned *period*);

**说明**

函数 **pulse** 用一个 DO 数据点产生一个指定周期的方波。

●*channel* 指定数字输出通道；

●*bit* 指定数字输出位

●*timer* 指定用以产生方波的定时器

●*on* 指定输出时间。

●*period* 指定波形周期。

如果发生错误，则产生一个任务错误码，如下所示。

TIMER_BADTIMER 定时器号无效

TIMER_BADVALUE 周期小于一个定时器周期

TIMER_BADADDR 数字通道无效

**注**

时钟的计数时间长度由函数 **interval** 设值。缺省值为 0.1 秒。

用函数 **settimer** 设置参数 *timer* 为 0，可以中止方波。如果控制器复位方波中断。

为了保证正确产生方波，应按下面的顺序执行。

```
settimer(t, 0);                    /*停止时钟*/
request_resource(IO_SYSTEM);
turnoff(c, b);                     /*方波以上升沿开始*/
release_resource(IO_SYSTEM);
pulse(c, b, t, o, p);              /*开始输出脉冲*/
```

如果指定的 I/O 点为 ON，方波序列以下降沿开始。

用函数 **timerout** 产生不规则的或不重复的序列。

可以使用符号 DIN_START, DIN_END, DOUT_START 和 DOUT_END 来指定数字通道。

**参考**

**pulse_train, settimer, timeout, ioWrite8Out**

**例子**

```
settmer(10,0);                     /*停止时间*/
request_resource(IO_SYSTEM);
turnoff(3,7);
release_resource(IO_SYSTEM);
interval(10,10);                   /*设置时钟比率为 1 秒*/
pulse(3, 7, 10, 3, 5);                 /* on = 60% */
```

# pulse_train

## 产生有限个脉冲

**语法**

#include “iohw.h”

void pulse_train(unsigned *channel*, unsigned *bit*, unsigned *timer*, unsigned *pulses*);

**说明**

函数 **pulse_train** 在一个数字输出点上产生一个脉冲数。输出是一个占空比为 50%周期是 200ms 的方波。

- *channel* 指定数字输出通道。
- *bit* 指定数字输出位。
- *timer* 指定产生方波的时钟。
    - *pulses* 指定脉冲数。用 *interval* 作为脉冲数量系数，产生的脉冲数量为 *pulse * interval*。

如果发生错误，则产生一个任务错误码，如下所示

TIMER_BADTIMER 定时器号无效

TIMER_BADVALUE 周期小于一个定时器周期

TIMER_BADADDR 数字通道无效

**注**

使用函数 **settimer** 把参数 *timer* 设为 0 可以中止方波。如果控制器复位方波中止。

为了保证正确产生方波，应按下面的顺序执行。

```
settimer(t, 0);                    /* 停止定时器 */
request_resource(IO_SYSTEM);
turnoff(c, b);                     /* 脉冲以上升沿开始 */
release_resource(IO_SYSTEM);
pulse_train(c, b, timer, pulses);
```

用函数 **timerout** 产生不规则的或不重复的序列。

可以使用 DIN_START, DIN_END, DOUT_START 和 DOUT_END 加上偏移来指定数字通道。

**参考**

**pulse, settimer, timeout, ioWrite8Dout**

**例子**

下面的程序段在通道 3、位 4 上产生 300 个脉冲。

```
interval(2, 1);
pulse_train(3, 4, 2, 300);
```

下面的程序段同样在通道 3、位 4 上产生 300 个脉冲。

```
interval(2, 100);
pulse_train(3, 4, 2, 300);
```

# queue_mode

## 控制传输数据

**语法**

```
#include “serial.h”
void queue_mode(FILE *stream, int mode);
```

**说明**

函数 queue_mode 控制串行数据的传输。正常情况下，输出到串口的数据被放在传输缓冲区中，当硬件准备好后即可被发送。

如果缓冲使能，数据就保存在缓冲区中直到缓冲被禁止。如果缓冲区满了，缓冲自动被禁止。

参数 stream 指定串口。如果无效则函数没有作用。

参数 mode 指定缓冲控制。它的值为 DISABLE、ENABLE。

# read_timer_info

## 得到时钟状态

**语法**

```
#include “iohw.h”
struct timer_info read_timer_info(unsigned timer);
```

**说明**

函数 **read_timer_info** 得到由参数 *timer* 指定的时钟的信息。

函数 **read_timer_info** 返回包含时钟信息的结构 timer_info。详细信息请参阅关于结构 timer_info 的详细描述。

**参考**

**settimer, timer, pulse_train, timeout**

**例子**

下面程序开始一个脉冲序列并显示时钟信息。

```
#include “iohw.h”
void main(void)
{
    struct timer_info tinfo;

    interval(10,1);
    pulse_train(3, 5, 10, 500);
    while(timer(10)>100)
    {
```

```
        release_processor();
    }
    tinfo=read_timer_info(10);
    printf("Pulses Remaining: %d\r\n", tinfo.time/2);
    printf("Output Channel: %d\r\n", tinfo.channel);
    printf("Output Bit : %d\r\n", tinfo.bit);
}
```

# receive_message

## 接收一个消息

**语法**

```
#include "primitiv.h"
envelope *receive_message(void);
```

**说明**

函数从当前任务消息队列中读取一个消息。如果队列为空，则任务终止直到有一个消息发给他。函数返回一指向结构 envelope 的指针。

**注**

关于信息的更多信息请参阅实时操作系统章节。

**参考**

**send_message, poll_message**

**例子**

下面例子等待并打印消息内容。

```
#include "primiriv.h"
#include "stdio.h"
void show_message(void)
{
    envelope *msg;
    while(TRUE)
    {
        msg=receive_message();
        printf("Message data %d\r\t", msg->data);
        deallocate_envelope(msg);
    }
}
```

# release_processor

## 释放处理器

**语法**

```
#include "primitiv.h"
void release_processor(void);
```

**说明**

函数 **release_processor** 释放对 CPU 的控制。其它具有相同优先级的任务可以运行。

**注**

程序中必须在一个空循环中调用此函数，允许执行其它任务。

在一个任务中释放控制器前应先释放所有资源。

关于任务和任务的时序安排的更多信息请参阅实时操作系统章节。

**参考**

**release_resource**

# release_resource

## 释放对资源的控制

**语法**

#include “primitiv.h”

void release_resource(int *resource*);

**说明**

函数 **release_resource** 释放参数 *resource* 指定的资源。

如果其它的任务正在等待这个资源，则将执行具有最高运算级的任务。如果没有其它等待的任务，则当前任务继续执行。

**注**

关于更多的资源信息请参阅实时操作系统章节。

**参考**

**request_resource, poll_resource**

**例子**

参阅函数 request_resource 的例子

# report_error

## 设任务错误码

**语法**

#include “primitiv.h”

void report_error(int error);

**说明**

函数 report_error 给当前任务设错误码，每一个当前正在执行的任务都维护一个错误码。

**注**

用于可共享 I/O 例程中，用来返回错误号到调用它的任务

**参考**

check_error

# request_resource

## 得到资源的控制

**语法**

#include “primitiv.h”

void request_resource(int *resource*);

**说明**

函数 **request_resource** 得到对参数 *resource* 指定的资源的控制。如果资源正在使用，则任务被锁定，直到资源可以被使用。

**注**

使用函数 **request_resource** 可访问非共享的资源。关于资源更多信息请参阅实时运算系统章节。

**参考**

**release_resource, poll_resource**

**例子**

下面代码段得到动态内存资源，分配并释放资源。

```
#include “primitiv.h”
#include “stdlib.h”
void task(void)
{
    unsigned *ptr;
    /*加入代码*/
    request_resource(DYNAMIC_MEMORY);
    ptr=(unsigned *)malloc((size_t)100);
    release_resource(DYNAMIC_MEMORY);
    /*加入代码*/
}
```

# resetClockAlarm

## 确认并复位实时时钟报警

**语法**

#include “rtc.h”

void resetClockAlarm(void);

**说明**

报警设置保留在实时时钟内。报警重新发生之前必须确认原来的报警。

函数 **resetClockAlarm** 确认上次实时时钟报警并重新使报警有效。如果通过报警唤醒后调用此函数报警将会出现次报警 24 小时后.。

**注**

此函数在实时时钟报警后被调用。包括从 **sleep** 函数返回码为 WS_REAL_TIME_CLOCK。

此函数不改变报警时间。

调用此函数前必须调用资源 IO_SYSTEM。

**参考**

**setClockAlarm, getClockAlarm, alarmIn**

**例子**

参考函数 installClockHandler 的例子

# route

## 重定向标准 I/O 流

**语法**

#include “serial.h”

void rout(FILE **logical*, FILE *hardware*);

**说明**

函数重定到 I/O 流到与 **stdout, stdin,stderr** 连接的设备中。这些流被定向到 **com1** 串口上。

参数 *logical* 指定要重定向的流。*hardware* 指定要输出数据的设备。它可以是 **com1** 或 **com2**.

**注**

这个函数会影响全局。

输出流必须重定向到一个支持输出的设备。输入流必须重定向到一个支持输入的设备。

**例子**

下面的例子重定向所有输入、输出、和错误到串口 **com2**。

```
#include “serial.h”

route(stderr,com2);
route(stdout,com2);
route(stdin,com2);
```

# runLed

## 控制 LED 的运行状态

**语法**

#include “iohw.h”

void runLed(unsigned *state*);

**说明**

函数 **runLed** 设定 **RUN LED** 为指定状态。参数 *state* 可以为下列值

LED_ON　　开 **LED**

LED_OFF　　关闭 **LED**

**RUN LED** 将保持当前状态值到状态被改变，或控制器复位。

**注**

梯形逻辑解释器器控制 RUN LED 的状态。如果控制器中有梯形逻辑，则 C 程序不应使用此函数。

**参考**

**error**

**例子**

```
#include “iohw.h”

void main(void)
{
    runLed(LED_ON);
    /*  程序其余部分  */
}
```

# send_message

## 给任务发送消息

**语法**

```
#include “primitiv.h”
void send_message(envelope *penv);
```

**说明**

函数 **send_message** 给任务发送一个消息。由参数 *penv* 指定的消息包包含了信息的目的任务、类型和数据。

此消息会加入目的任务的消息序列中。如果目的任务正在等待一个消息，那么此任务将开始执行该消息。

函数 **send_message** 不会使当前任务锁定。

**注**

函数 **allocate_envelope** 可以得到此消息包。

**参考**

**receive_message, poll_message, allocate_envelope**

**例子**

下面程序创建一个任务，用来显示和发送消息。

```
#include “stdio.h”
#include “primitiv.h”

void show(void)
```

```
{
    envelope *msg;

    while(TRUE)
    {
        msg=receive_message();
        printf("Message data %d\r\t",msg->data);
        deallocate_envelope(msg);
    }
}

void main(void)
{
    envelope *msg;
    unsigned tid;

    tid=create_task(showIt, 2, APPLICATION, 1);
    msg=allocate_envelope();
    msg->destination=tid;
    msg->type=MSG_DATA;
    msg->data=1002;
    send_message(msg);

    while(TRUE)
    {
            release_processor();
    }
}
```

# setBootType

## 设置控制器启动方式

**语法**

#include "system.h"

void setBootType(unsigned *type*);

**说明**

函数 **setBootType** 定义控制器启动类型。系统启动程序使用此函数。应用程序不能使用此函数。

**注**

此函数设的值可以被函数 **getBootType** 读取。

# setclock

## 设置实时时钟

**语法**

#include “rtc.h”

void setclock(struct clock **now*);

**说明**

函数 **setclock** 设置实时时钟。参数 *now* 包含了要设值的时间和日期。

参考 rtc.h 中结构 clock 的描述。为了使实时时钟正常工作，结构 clock 的属性必须设为有效值。

**参考**

**getclock**

**例子**

```
clude “rtc.h”
#include “primitiv.h”

void daylight(void)
{
    struct clock now;

    request_resource(IO_SYSTEM);
    now=getclock();
    now.hour=now.hour+1%24;
    setclock(now);
    request_resource(IO_SYSTEM);
}
```

# setClockAlarm

## 设置实时时钟报警

**语法**

#include “rtc.h”

unsigned setClockAlarm(ALARM_SETTING *alarm*);

**说明**

函数 **setClockAlarm** 设定实时时钟在某一时刻报警。参数 *alarm* 指定报警的时间。

关于结构 ALARM_SETTING 的详细信息请参阅关于 rtc.h 的描述。

如果设置正确，函数返回 TRUE,否则返回 FALSE。如果设值发生错误，则原来的报警设置不会改变。

**注**

报警只会发生一次，但他的设置不变，直到设置变为无效。可以使用函数 **resetClockAlarm** 函数确认已经发生的报警，并重新使报警设置有效。

报警类型设为 AT_NONE 可以使报警无效，此时不必指定小时、分钟和秒。

调用此函数前必须申请资源 IO_SYSTEM

**参考**

**alarmIn, getClockAlarm, resetClockAlarm, installClockHandler**

**例子**

```
#include “rtc.h”
#include “primitiv.h”
#include “system.h”
void wakeUpAtEight(void)
{
    ALARM_SETTING alarm;
    Unsigned wakeSource;

    alarm.type=AT_ABSOLUTE;
    alarm.hour=8;
    alarm.minute=0;
    alarm.second=0;

    request_resource(IO_SYSTEM);
    setClockAlarm(alarm);
    release_resource(IO_SYSTEM);

    do
    {
            request_resource(IO_SYSTEM);
            wakeSource=sleep();
            release_resource(IO_SYSTEM);
    }until(wakeSource == WS_REAL_TIME_CLOCK);

    alarm.type=AT_NONE;
    request_resource(IO_SYSTEM);
    setClockAlarm(alarm);
    release_resource(IO_SYSTEM);
}
```

# setdbase

## 向 I/O 数据库写数据

**语法**

```
#include “database.h”
```

void setdbase(unsigned *type*, unsigned *address*, int *value*);

**说明**

函数 **setdbase** 向数据库写数据。参数 *type* 指定数据库寻址方法。*address* 指定地址。地址类型和范围如下。

| 类型 | 地址范围 | 寄存器长度 |
|---|---|---|
| MODBUS | 00001 到 NUMCOIL<br>10001 到 10000+NUMSTATUS<br>30001 到 30000+NUMINPUT<br>40001 到 40000+NUMHOLDING | 1 比特<br>1 比特<br>16 比特<br>16 比特 |
| LINEAR | 0 到 NUMLINEAR-1 | 16 比特 |

**注**

如果指定的寄存器正处于强制状态，则数据库保持不变。

当写 LINEAR 数字 I/O 地址时，参数 *value* 一次写入 16 位位寄存器中。只有当前被强制的寄存器的值保持不变。

控制器复位不会改变 I/O 数据库的值。他是一个永久存储区。

更多信息请参阅 I/O 数据库和寄存器分配的有关章节。

调用此函数前必须先申请资源 IO_SYSTEM。

**参考**

**dbase, overrideDbase, setForceFlag**

**例子**

```
#include “primitiv.h”
#include “database.h”
void main(void)
{
    request_resource(IO_SYSTEM);

    setdbase(MODBUS, 40001, 102);
    setdbase(LINEAR, START_COIL, 255);
    setdbase(LINEAR, 3020, 240);
    setdbase(LINEAR, START_HOLDING, 330);
    setdbase(LINEAR, START_HOLDING, 330);

    release_resource(IO_SYSTEM);
}
```

# setDTR

## 控制 RS232 端口 DTR 信号

**语法**

#include “serial.h”
void setDTR(FILE *_stream_, unsigned _state_);

**说明**

函数 **setDTR** 设置由参数 *stream* 指定的通讯端口的 DTR 信号线的状态。

当状态 *state* 位 SIGNAL_ON,表示信号线被置有效。当状态 *state* 位 SIGNAL_OFF, 表示信号线无效。

**注**

DTR 信号线遵循正常的 RS232 电压规范。

此函数只对 RS232 端口有用。

# setForceFlag

## 设置寄存器强制标志状态

**语法**

#include “database.h”
unsigned setForceFlag(unsigned *type*, unsigned *address*, unsigned *value*);

**说明**

函数 **setForceFlag** 为指定的数据库寄存器设置强制标志值 *value*。参数 *value* 取 0 或 1，或对 LINEAR 数字地址设置一个 16 比特掩码数据。

如果 *address* 或地址类型 *type* 无效，则强制标志不变，函数返回 FALSE。其它情况返回 TRUE。

有效地值和类型如下表。

| 类型 | 地址范围 | 寄存器长度 |
|---|---|---|
| MODBUS | 00001 到 NUMCOIL<br>10001 到 10000+NUMSTATUS<br>30001 到 30000+NUMINPUT<br>40001 到 40000+NUMHOLDING | 1 比特<br>1 比特<br>16 比特<br>16 比特 |
| LINEAR | 0 到 NUMLINEAR-1 | 16 比特 |

**注**

如果寄存器强制标志被设定，那么 IO 数据库的值被强制到当前的值。这时此寄存器得只有用函数 **overridDbase** 或是用 ELadder 中编辑/强制寄存器对话窗可以改变。强制期间函数 **setdbase**, protocol 或梯形逻辑程序都不能改变此值。

控制器复位不改变强制标志。强制标志位于永久存储区。

更多信息请参阅 IO 数据库和寄存器分配的有关章节。

调用此函数前必须先申请资源 IO_SYSTEM。

**参考**

**getForceFlag, clearAllForcing, overrideDbase**

**例子**

```
#include “primitiv.h”
#include “database.h”

void main(void)
{
    request_resource(IO_SYSTEM);

    setForceFlag(MODBUS, 40001, 0);
    setForceFlag(LINEAR, 302, 255);

    release_resource(IO_SYSTEM);
}
```

# setjiffy

## 设置瞬时时钟

**语法**

```
#include “rtc.h”
void setjiffy(unsigned long value);
```

**说明**

函数 **setjiffy** 设置系统瞬时时钟。瞬时时钟每 1/60 秒递加一次。瞬时时钟计数到 5183999 后翻转到 0。

**注**

实时时钟和瞬时时钟没有联系。他们可能会有很小的偏差。使用瞬时时钟计时比用 1/10 秒定时器精度高。

**参考**

**timer, interval, jiffy**

**例子**

参看函数 jiffy 的例子

# setOutputsInStopMode

## 设置停止模式输出

**语法**

```
#include “rassign.h”
void setOutputsInStopMode(unsigned doutsInStopMode, unsigned aoutsInStopMode);
```

**说明**

函数 **setOutputsInStopMode** 设置 *doutsInStopMode* 和 *aoutsInStopMode* 控制标志到指定状态。

如果 *doutsInStopMode* 设为 TRUE，那么当梯形图程序停止后数字输出将保持原来状态。如果 *doutsInStopMode* 设为 FALSE，那么当梯形图程序停止后数字输出被关闭。

如果 *aoutsInStopMode* 设为 TRUE，那么当梯形图程序停止后模拟输出将保持。如果 *aoutsInStopMode* 设为 FALSE，那么当梯形图程序停止后模拟输出变为零。

**参考**

**getOutputsInStopMode**

**例子**

```
#include “mriext.h”
#include “rassign.h”

void main(void)
{
    unsigned holdDoutsOnStop;
    unsigned holdAoutsOnStop;

    getOutputsInStopMode(&holdDoutsStop, &holdAoutsOnStop);
    holdAoutsOnStop=TRUE;
    setOutputsInStopMode(holdDoutsOnStop, holdAoutsOnStop);
}
```

# set_pid

## 设置 PID

**语法**

```
#include “pid.h”
void set_pid(unsigned name, unsigned block, int value);
```

**说明**

函数 **set_pid** 设置 PID 各个控制块参数。参数 *name* 必须为 pid.h 中的一个变量名(宏定义)。*Block* 必须在 **0～PID_BLOCKS-1** 范围之内。

**注**

如果需要详细了解 PID 控制，可以参阅**《ELadder PID 使用手册》**，

PID 控制块的值在程序运行时不会被初始化。并且在掉电、程序装载的时候不会改变。PID 控制块需要用户程序初始化。

调用此函数前必须先申请资源 IO_SYSTEM。

**参考**

**get_pid，auto_pid，clear_pid**

# set_port

## 设置串口

**语法**

#include “serial.h”

void set_port(FILE *_stream_, struct pconfig *_settings_);

**说明**

函数 set_port 设置通讯端口参数。参数 stream 必须取位 com1 或 com2。参数 settings 表示串口配置结构。参考结构 pconfig。

**注**

如果设置和当前串口相同则函数什么也不做。

设置改变时串口复位。此时收到的数据和缓存中的数据被丢弃。

调用此函数前必须先调用资源 IO_SYSTEM。

**参考**

**get_port**

**例子**

下面代码段改变 COM2 的波特率

```
#include “serial.h”
#include “primitiv.h”
struct pconfg settings;

get_port(com2, &settings);
settings.baud=BAUD19200;
request_resource(IO_SYSTEM);
set_port(com2, &settings);
release_resource(IO_SYSTEM);
```

下面代码段把 com2 设成和 com1 相同的配置。

```
#include “serial.h”
#include “primitiv.h”
struct pconfig settings;

request_resource(IO_SYSTEM);
set_port(com2, get_port(com1, &settings));
release_resource(IO_SYSTEM);
```

# setProgamStatus

## 设置程序状态标志

**语法**

#include “system.h”

void setProgramStatus(unsigned *status*);

**说明**

函数 **setProgramStatus** 设置应用程序状态标志。当控制器冷启动或一个程序从程序装载下载到控制器中时状态标志被设为 NEW_PROGRAM。

**注**

此标志有两个预定义值。

NEW_PROGRAM　　指明程序是新下载的

PROGRAM_EXECUTED　指明程序已经执行

**参考**

**getProgramStatus**

**例子**

参考函数 getProgramStatus 的例子

# set_protocl

## 通讯端口设置

**语法**

#include “protocol.h”

int set_protocol(FILE *_stream_, struct prot_settings *_settings_);

**说明**

函数 **set_protocol** 设置协议参数。参数 *stream* 必须为 *com1* 或 *com2*。*settings* 是一个协议结构。参阅关于结构 prot_settings 的描述。

如果设置改变，函数返回 TRUE，否则返回 FALSE。

调用此函数前必须先申请资源 IO_SYSTEM。

**注**

设置协议类型位 NO_PROTOCOL，将结束协议任务，并释放分配给他的栈资源。

**参考**

**get_protocol, start_protocol, get_protocol_status**

**例子**

下面的程序段改变 com2 状态号到 4。

```
#include “protocol.h”
#include “primitiv.h”
struct prot_settings settings;

get_protocol(com2, &settings);
settings.station=4;
request_resource(IO_SYSTEM);
set_protocol(com2, &settings);
release_resource(IO_SYSTEM);
```

# setStatus

## 设置控制器状态码

**语法**

#include “iohw.h”

void setStatus(unsigned *code*);

**说明**

函数 setStatus 设置设置控制器状态码。当状态码不为零，状态灯按照状态码的二进制序列闪烁。如果 code 为零，状态灯关闭。

**注**

如果 *code* 非零，状态输出打开。更多信息参阅系统硬件设备手册。

二进制序列包含了指示错误的 Status LED 的长短闪烁。二进制零表示短的闪烁（1/10 秒），二进制一表示长的闪烁（1/2 秒）。首先输出低位。高位的零被忽略不计。每次显示重复之间有两秒的延迟。

状态码的 0 位和 1 位用于寄存器分配。推荐使用函数 setStatusBit 以防止改变这些位。

**参考**

**setStatusBit, clearStatusBit, getStatusBit**

# setStatusBit

## 设置控制器状态码的比特位

**语法**

#include “iohw.h”

unsigned setStatusBit(unsigned *bitMask*);

**说明**

函数 **setStatusBit** 设置由参数 *bitMask* 指定的状态码的比特位。当状态码不为零，

**状态灯**按照状态码的二进制序列闪烁。如果 *code* 为零，状态灯关闭。

函数返回状态寄存器的值。

**注**

如果 *code* 不为零，状态输出打开。更多信息请参考系统设备手册。

二进制序列包含了指示错误的 LED 的长短闪烁。二进制零表示短的闪烁（1/10 秒），二进制一表示长的闪烁（1/2 秒）。首先输出低位。开头的零被忽略不计。每次重复之间有两秒的延迟。

状态码 0 位和 1 位用于寄存器分配。

**参考**

**clearStatusBit, getStatusBit, setStatus**

# settimer

## 设置一个定时器

**语法**

#include “iohw.h”

void settimer(unsigned *timer*, unsigned *value*);

**说明**

函数 **settimer** 把参数 *value* 的值赋给定时器。定时器按照时钟频率递减计数。

函数 **settimer** 可重置没完成计数的定时器。

**参考**

interval

**例子**

下面代码段设定定时器 8 为 10 秒定时，0.5 秒间隔

interval(8, 5);/*间隔 0.5 秒*/

settimer(8, 20);/*10 秒钟*/

下面代码段设定定时器 9 为 60 秒定时，1 秒间隔

interval(9, 10);/*间隔 1 秒*/

settimer(9, 60);/*60 秒钟*/

# setWakeSource

## 设置从睡眠模式唤醒条件

**语法**

#include “system.h”

void setWakeSource(unsigned *enableMask*);

**说明**

**setWakeSource** 例行程序使唤醒处理器的资源有效或无效。他使所有由参数 *enableMask* 指定的唤醒源为有效，其它唤醒源无效。

有效的唤醒源如下表，可以指定多个唤醒源。

- WS_NONE
- WS_ALL
- WS_REAL_TIME_CLOCK
- WS_INTERRUPT_INPUT
- WS_LED_POWER_SWITCH
- WS_COUNTER_0_OVERFLOW
- WS_COUNTER_1_OVERFLOW
- WS_COUNTER_2_OVERFLOW

**注**

唤醒资源 WS_NONE 防止控制器从唤醒转向复位，除非掉电。

**参考**

**getWakeSource, sleep**

**例子**

下面代码段显示如何使唤醒源有效或无效。

```
/*唤醒所有资源*/
setWakeSource(WS_ALL);

/*仅使实时时钟有效*/
setWakeSource(WS_REAL_TIME_CLOCK);
```

# signal_event

## 事件发生信号

**语法**

#include “primitiv.h”

void signal_even(int *event_number*);

**说明**

函数 **signal_even** 用通知 *event_number* 事件发生了。

如果有任务等待此事件，那么最高级别的任务准备执行，否则事件标志递增。可以记录总共 255 个事件。如果有更高级别的任务等待事件，则当前任务被阻塞。

**注**

关于事件的更多信息请参阅时实操作系统的有关章节。

有效设事件号为：0 到 RTOS_EVENTS-1。

**参考**

**wait_event, poll_event, interrupt_signal_event**

**例子**

```
#include “stdio.h”
#include “mriext.h”
#include “primitiv.h”
#include “iohw.h”

void task1(void)
{
    while(TRUE)
    {
        wait_event(20);
        printf(“Event 20 occurred\r\t”);
    }
```

```
}

void main(void)
{
    create_task(task1, 3, APPLICATION, 4);

    while(TRUE)
    {
        settimer(0, 10);/*间隔 1 秒*/
        while(timer(0))/*等待 1 秒*/
        {
            /*允许其它程序执行*/
            release_processor();
        }
    }
    signal_event(20);
}
```

# sleep

## 控制器睡眠

**语法**

```
#include “system.h”
unsigned sleep(void);
```

**说明**

函数 **sleep** 使控制器进入睡眠模式。电源消耗减少到很小，所有程序被中止，直到控制器恢复运行。睡眠模式下所有输出被关闭。

控制器在下表列出的条件下恢复执行。应用程序可以使一些唤醒条件无效。如果一个唤醒条件无效，那么当此条件发生时控制器不会恢复执行。

| 条件 | 唤醒影响 | 允许无效 | 无效时影响 |
|---|---|---|---|
| 设备复位 | 应用程序从头执行。 | 否 | 不适用 |
| 外部中断 | 程序从睡眠点继续执行。 | 是 | 中断输入被忽略 |
| 实时时钟报警 | 程序从睡眠点继续执行。 | 是 | 报警被忽略 |
| LED 电源按钮 | 程序从睡眠点继续执行。 | 是 | LED 电源按钮被忽略 |

| | | | |
|---|---|---|---|
| 硬件计数翻转 | 程序从睡眠点继续执行<br>程序从睡眠点继续执行。 | 是 | 程序从睡眠点继续执行。控制器转向睡眠模式。 |

函数 sleep 返回唤醒码，此唤醒码指示的条件使控制器重新执行。

| 返回码 | 条件 |
|---|---|
| WS_REAL_TIME_CLOCK | 实时时钟报警 |
| WS_INTERRUP_INPUT | 中断输入向上脉冲 |
| WS_LED_POWER_SWITCH | LED 电源开关按下 |
| WS_COUNTER_0_OVERFLOW | 计数器 0 低字溢出（每 65536 溢出一次） |
| WS_COUNTER_1_OVERFLOW | 计数器 1 低字节溢出（每 65536 溢出一次） |
| WS_COUNTER_2_OVERFLOW | 计数器 2 低字节溢出（每 65536 溢出一次） |

调用此函数前必须先申请资源 IO_SYSTEM。

**参考**

setClockAlarm, alarmIn, setWakeSource, getWakeSource

**例子**

参考关于函数 setClockAlarm 和 alarm 的例子。

# start_protocol

## 使协议任务有效

**语法**

#include “protocol.h”

int start_protocol(FILE **stream*);

**说明**

函数 **start_protocol** 使由参数 *stream* 指定的端口上的协议任务有效。

如果协议启动，函数返回 TRUE，否则返回 FALSE。

**注**

函数 **start_protocol** 被用于系统启动程序。应用程序应使用函数 **set_protocol** 控制协议

运行。

**参考**

**set_protocol, get_protocol**

# startup_task

## 任务启动

**语法**

#include “system.h”

void *startup_task(void);

**说明**

函数 **starup_task** 返回系统或应用任务启动地址。

**注**

函数用于复位程序。他一般不被用于应用程序中。

# startTimedEvent

## 允许正规事件发信号

**语法**

#include “system.h”

unsigned startTimedEvent(unsigned *event*, unsigned *interval*);

**说明**

函数 **startTimedEvent** 使得由参数 *event* 指定的事件按指定间隔 *interval* 发生信号。参数 *interval* 的度量单位是 0.1 秒。接收事件的任务使用函数 **wait_event** 或 **poll_event** 来侦测事件。

如果事件可以产生，函数返回 TRUE。如果间隔 interval 为零或事件号无效函数返回 FALSE，

并且上一个被使能的事件不变。

**注**

有效的事件号为 0 到 RTOS_EVENTS-1。在 primitiv.h 中定义的事件不能被用在用户应用程序中。

当等待事件的任务结束后，应用程序应停止定时事件信号的发送。如果事件信号没有停止发送，那么事件将被加到队列中，直到有函数又在等待它。

**参考**

endTimedEvent, signal_event, wait_event, poll_event

**例子**

下面的例子每隔 10 秒打印一次时间。

#include “string.h”

```
#include “mriext.h”
#include “serial.h”
#include “protocol.h”
#include “primitiv.h”
#include “rtc.h”
#define TIME_TO_PRINT    15
/*-----------------------------------------------------------------
shutdown 函数停止 TIME_TO_PRINT 事件的信号发送
-----------------------------------------------------------------*/
void shutdown(void)
{
      endTimedEvent(TIME_TO_PRINT);
}

void main(void)
{
      struct prot_settings settings;
      struct clock now;
      TASKINFO taskStatus;

      settings.type=NO_PROTOCOL;
      settings.station=1;
      settings.priority=3;
      settings.SFMessaging=FALSE;
      request_resource(IO_SYSTEM);
      set_protocol(com1, &settings);
      release_resource(IO_SYSTEM);

      taskStatus=getTaskInfo(0);
      installExitHandler(taskStatus.tasked, shutdown);

      startTimedevent(TIME_TO_PRINT);

      while(TRUE)
      {
            wait_event(TIME_TO_PRINT);
            request_resource(IO_SYSTEM);
            now=getclock();
            release_resource(IO_SYSTEM);
            fprintf(com1, “Time %02u:%02u:%02u:\r\t”, now.hour, now.minute, now.second);
      }
}
```

# timeout

## 延迟数字输出

**语法**

#include “iohw.h”

void timeout(unsigned *channel*, unsigned *bit*, unsigned *timer*, unsigned *delay*);

**说明**

函数 **timeout** 开始一个数字输出延时操作。

●*channel* 指定数字输出通道。

●*bit* 指定 *channel* 中的输出点。

●*timer* 指示延时的时间。他的范围是 0 到 31。

●*delay* 指示定时器延时的滴答数。函数 **interval** 设置时钟一次滴答的时间长度。

如果发生一个错误，则任务错误码被设置如下：

TIMER_BADTIMER　如果定时器号无效

TIMER_BADADDR　如果数字通道或比特无效。

**注**

要取消暂停，设置定时器为零。

使用函数 **pulse** 产生一个连续的方波序列。

函数 **timeout** 可以在上一个结束之前开始一个新的 timeout 序列。这种情况下上一个将结束，新的开始。

可以使用寄存器操作来替代此函数。

**参考**

**interval, turnon, turnoff, settimer, pulse**

# timer

## 读定时器

**语法**

#include “iohw.h”

unsigned timer(unsigned *timer*);

**说明**

函数 **timer** 返回定时器 *timer* 中剩余时间。参数 *timer* 范围时 0 到 31。返回值为零表示定时器已经计数完毕。

如果定时器号无效，函数值返回零并且任务错误码被设置为 TIMER_BADTIMER。

**参考**

**interval, timeout, settimer, pulse, read_timer_info**

**例子**

下面代码段设置一个定时器，并在他达到零之前显示时间。

```
#include “stdio.h”
#include “iohw.h”
interval(0, 1);
settimer(0, 10);
while(timer(0))
    printf(“Timer %d\r\t”,timer(0));
```

# wait_event

## 等待事件

**语法**

```
#include “primitiv.h”
void wait_event(int event);
```

**说明**

函数 **wait_event** 检测一个事件是否发生了。如果这个事件发生了，则事件计数器减一，并且函数返回。如果事件没有发生，则任务被阻塞直到事件发生。

**注**

关于事件的更多信息青参阅实时操作系统的有关章节。

有效的事件号为 0 到 RTOS_EVENT-1。在应用程序中不能使用在 primitiv.h 中定义的事件号。

**参考**

**signal_event, poll_event, startTimedEvent**

**例子**

参考函数 signal_event 的例子。

# wd_auto

## 自动看门狗时钟模式

**语法**

```
#include “iohw.h”
void wd_auto(void);
```

**说明**

函数 **wd_auto** 把对看门狗的控制交给操作系统。定时器由系统自动更新。

**注**

更多信息请参考有关看门狗章节。

**参考**

**wd_manual**, **wd_pulse**

**例子**

请参考函数 **wd_manual** 函数的例子。

# wd_manual

## 手动控制看门狗模式

**语法**

#include “iohw.h”

void wd_manual(void);

**说明**

函数 wd_manual 控制看门狗。

**注**

应用程序必须至少每隔 0.5 秒使用函数 wd_pulse 触发看门狗时钟，以防止控制器复位。

更多信息请参考有关看门狗章节。

**参考**

**wd_auto**, **wd_pulse**

**例子**

```
#include “iohw.h”
void main(void)
{
    wd_manual();
    wd_pulse();
    /*…代码执行时间小于 0.5 秒*/
    wd_pulse();
    /*…代码执行时间小于 0.5 秒*/
    ws_auto();
}
```

# wd_pulse

## 重新触发看门狗定时器

**语法**

#include “iohw.h”

void wd_pulse(void);

**说明**

函数 **wd_pulse** 重新触发看门狗定时器。

**注**

如果已经执行了函数 **wd_manual**，必须至少每隔 0.5 秒使用函数 **wd_pulse** 触发看门狗时钟，以防止控制器复位。

更多信息请参考有关看门狗章节。

**参考**

**wd_auto, wd_manual**

**例子**

请参阅函数 **wd_manual** 的例子。

# 第七章 C 程序开发

## 7.1 程序结构

EMC 支持单一文件和多个模块程序。应用程序包含由 C 提供的支持函数、主函数与用户写的其它函数。

C 应用程序可以包含于一个单一文件或称为模块的许多独立文件中。单一文件容易编译和连接。编辑和编译的难度随文件的大小而改变。

当只有一些模块改变时，存储于独立模块的应用更容易编辑和编译。被编译的模块可合并成一个目标库，在用户中共享。

### 7.1.1 主函数结构

下面为一个典型的主函数结构程序举例。

```
void main(void)
{
    while (TRUE)
    {
    }
}
```

初始化动作包括变量说明、变量初始化与当程序开始运行时必须执行的以前的动作。

在程序的主循环前产生支持任务。程序运行期间任务可动态的生成与结束。程序主循环为一个连续执行动作的死循环。主函数通常不返回值。

**举例：**

下面为三个模块程序的举例。每一个函数存储在一个单独文件中。

File:func1.c

```
    #include “serial.h”

    void func1(void)
    {
        fputs("This si function 1\r\n",com1);
    }
```

Fele:func2.c

```
    #include “serial.h”

    void func2(void)
    {
        fputs("This sifunction 2\r\n",com1);
    }
```

File:main.c

```
#include “mriext.h”

extern void func1(void);
extern void func2(void);

void main (void)
{
    func1();

    while (TRUE)
    {
        func2();
    }
}
```

## 7.1.2 启动函数结构

用户的主函数从 C 的 appstart()函数调用。appstart()函数执行许多由用户更改的有用函数。

启动代码有 5 种主要功能：

- 建立和初始化应用程序堆
- 指定分配到主程序的堆栈模块数量
- 初始化应用程序变量
- 控制协议、梯形逻辑与后台 I/O 任务的执行
- 运行主程序

EMC 提供了启动函数的源代码。名称为 appstart.c

堆栈是由动态内存分配函数使用的一部分内存。堆栈开始于 RAM 的末端，直到物理 RAM 的末端。界限由下面的语句设置。

```
end-of-heap     .equ      41ffffh
```

界限由控制器默认的最小内存选项设置。如控制器有更多内存，根据下表改变常数值。

| 安装的 RAM | C 应用程序 RAM 地址 |
|---|---|
| 128K 字节 | 无 |
| 256K 字节 | 400000H-41FFFFh |
| 640K 字节 | 400000H-47FFFFh |
| 1024K 字节 | 388000H-3E7FFFh<br>400000H-47FFFFh |

文件中应用程序信号部分包含一个常数，此常数决定分配给主任务的堆栈容量。堆栈容量对多数应用是够用的。它可通过改变下面语句进行更改：

```
.word    4        ;stack    size    in    block
```

appstart()函数开始时进行初始化堆栈指针、设置所有未初始化的参数为 0 并初始化系统变量。然后根据存储在 EEPROM 的变量为每一个串口启动通讯协议。若应用程序从未使用通讯协议，则可以去除下面某些或所有命令，为协议命令释放堆栈空间。

```
start-protocol(com1)
start-protocol(com2)
start-protocol(com3)
start-protocol(com4)
```

后台 I/O 任务用于定时器函数、拨号 MODEM 通讯与 PID 控制器函数。若不使用这些函数，可改变下面语句中的 TRUE 为 FALSE，以减少 CPU 负荷 ：

```
runBackgroundIO(TRUE);
```

梯形逻辑注释器用于梯形逻辑中。若不使用梯形逻辑，可改变下面语句中的 TRUE 为 FALSE，以减少 CPU 负荷 ：

```
runladderlogic(TRUE);
```

最后运行主函数。-initcopy 函数从程序中的-INITDATA 部分拷贝初始化值到初始化变量。若数据中无错误便可运行用户的应用程序。

```
if (-initcopy()==0)
{
  main();
}
```

若主函数返回值，则任务结束。这使所有其它由主函数产生的应用任务停止。

### 7.1.3 数据存储

所有未被初始化的变量在程序启动时初始化为 0。I/O 数据库是内存中的一部分。存储在 I/O 数据库中的数据当电源掉电时仍保留其值。

多数情况中，I/O 数据库为数据存储提供了足够的空间。但若需要附加的未初始化内存，如一列自定义的数据结构，可按下面例子建立非初始化内存部分。

```
#include “datalog.h”
```

```
#include “datalog.h”
#pragma option -NZsavedata
#pragma option -Xp

unsigned logIndex;
struct dataLog logDSata[DATA_LOG_SIZE];
```

文件 datalog.c 中的任何变量可置于称为 savedata 的未初始化部分中。变量的操作码应置于一个单独文件中。此文件通过置于头文件的外部定义访问变量。

## 7.2 控制器初始化

当开始一个新的程序工程或从默认状态下启动时，应对控制器的内存进行初始化。程序

下载前，无需初始化控制器。

首先运行 SERVICE 引导程序。SERVICE 引导程序将程序和数据保存到静态 RAM 中，但不启动程序的运行。使用默认通讯参数。

运行 SERVICE 引导程序：

- 将电源从控制器移开。
- 按下 LED POWER 开关。
- 为控制器提供电源。
- 等待约 5 秒。
- 松开 LED POWER 开关。

第二，对控制器的程序与数据内存进行初始化。新控制器需要进行所有初始化。控制器初始化方法参见《ELadder 使用说明》。

## 7.3 下载程序

C 程序下载器对话框将可执行文件从 PC 机发送到控制器中。

下载程序到 RAM 中：

- 初始化控制器。
- 下载程序到控制器中。
- 运行 EMC。
- 从通讯菜单下，选择 C 程序下载命令。
- 进入 C 程序文件名编辑对话框中的可执行文件。
- 选择 C 程序写选项和其它需要的写选项。
- 点击写按钮。程序将被下载到 RAM 或 Flash。

为 C 程序产生一个校验和。程序每运行一次，校验和核对一次，防止程序运行发生错误。

## 7.4 执行程序

当从 EMC 程序下载器接收到运行程序命令或为控制器供电时，执行 C 程序。

从程序下载器启动程序：

- 运行 EMC 程序。
- 从 C 程序下载器对话框点击运行按钮执行程序。控制器可在 RAM 或 FLASH 中运行程序。

控制器按下面次序运行程序：

- RAM 中的 C 应用程序。
- FLASH 中的 C 应用程序。

- 没有 C 应用程序。

此运行方式在下面情况中非常有用。控制器安装了带有程序的 FLASH。若需要增加新的部件或进行改正，则程序可从本地或远程下载到 RAM 中。

若 RAM 中的程序丢失或损坏，则运行 FLASH 中的程序。

## 7.5 C 程序开发举例

程序开发包括三步骤：编辑、编译与连接、下载与运行。

通常，新的 C 编译器中运行的第一个程序为“hello,world”。打印的信息为“hello,world”。下面就以此程序为例简述 C 程序开发的步骤。

### 7.5.1 编辑

在文本模式下，可使用任意文本编辑器或文字处理器写程序。使用文本编辑器打开文件“hello.c”文件。它在\SEMC\路径下。程序与“hello,world”有些不同。

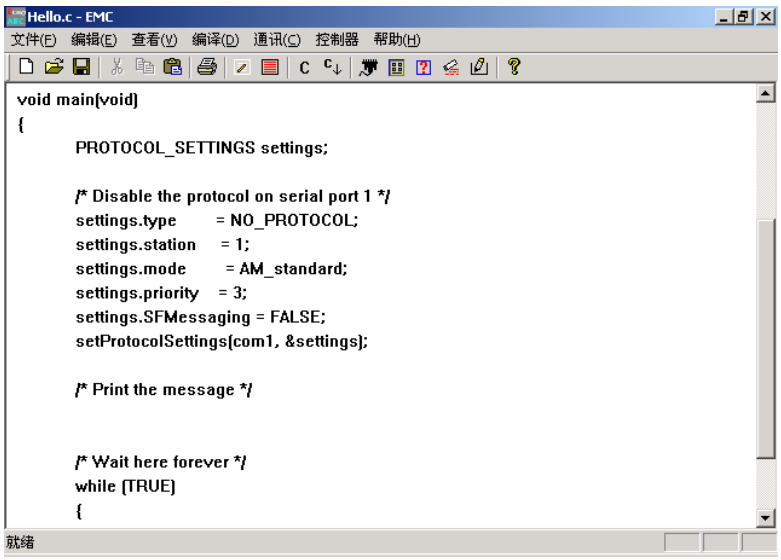



图 6-1

```
/*hello.c */
#include “primitiv.h”
#include “mriext.h”
#include “serial.h”
#include “protocol.h”
void main(void)
{
    struct prot-settings settings;

    //禁止串口 1 的协议。
    settings.type          =NO-PROTOCOL;
    settings.station       =1;
    settings.priority      =3;
    settings.SFMessaging =FALSE;
```

```
    requst-resouce(IO-SYSTEM);

    //打印信息
    fprintf(com1,“hello,world\r\n”);

    //等待
    while(true)
    {
        //允许执行其它任务
        release-processor();
    }
}
```

“hello,world”信息从控制器的串口 COM1 输出。连接在串口的终端显示此信息。

控制器通常使用 Modbus 通讯协议在所有端口上通讯。程序的第一段禁止 COM1 口协议，因此串口可以用作通用的 RS232 端口。

fprintf 函数打印信息到 com1 串口中。

当完成程序测试后，关闭 hello.c 文件，便可进行编译与连接。

## 7.5.2 编译与连接

编译与连接将源代码转换为控制器可执行代码。EMC 产生一个优化的代码。编译器与连接器运行于 MICROSOFT MS-DOS 运行系统下。

举例：

hello.c 程序在 EMC\目录下。编译并连接此程序：

- 在相应目录找到并打开 hello.c 程序；
- 点击编译菜单下的编译命令或工具栏中的 C 图标，可选择 RAM 或 Flash（默认为 RAM），代表编译连接完后生成的 ABS 可执行文件将下载到相应的 RAM 或 Flash 存储器。

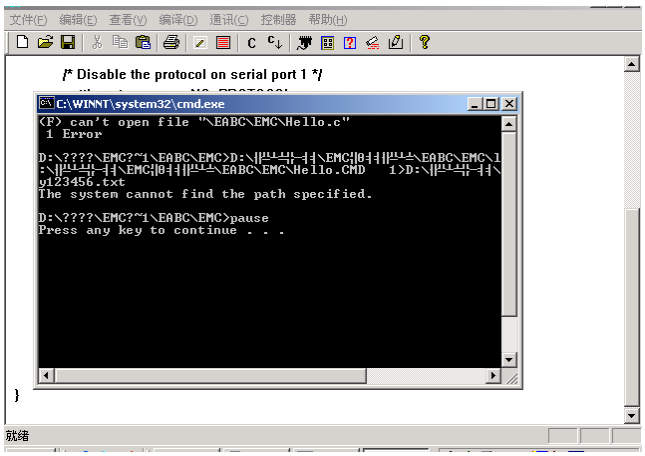



图 6-2

编译文件 hello.abs 包含可下载到控制器中的格式化的代码。

## 7.5.3 下载与运行

EMC 程序下载器将可执行代码从 PC 传递到控制器中，并控制程序的执行。下载器可初始化程序内存和串口配置。

### ➢ 控制器初始化

当开始一个新工程时需对控制器的存储器进行初始化。当下载每个程序时无需对控制器进行初始化。

首先执行 SERVICE 引导程序。SERVICE 引导程序将程序与数据保存到静态 RAM 中，但不启动程序运行。使用默认的通讯参数。

执行 SERVICE 引导程序：

- 断开控制器电源。
- 按住复位按钮的同时给控制器上电。
- 等待直到控制器第二个指示状态灯亮。
- 松开复位按钮。

接着初始化控制器中程序与数据存储器。

- 选择适当的串行电缆将 PC 连接到控制器。控制器 COM1、COM2 以及扩展的 COM 口均可用。
- 从通讯菜单中选择 PC 端口设置命令或工具栏的图标。

图 6-3

- 从通讯菜单中选择控制器类型设置命令或工具栏的图标。

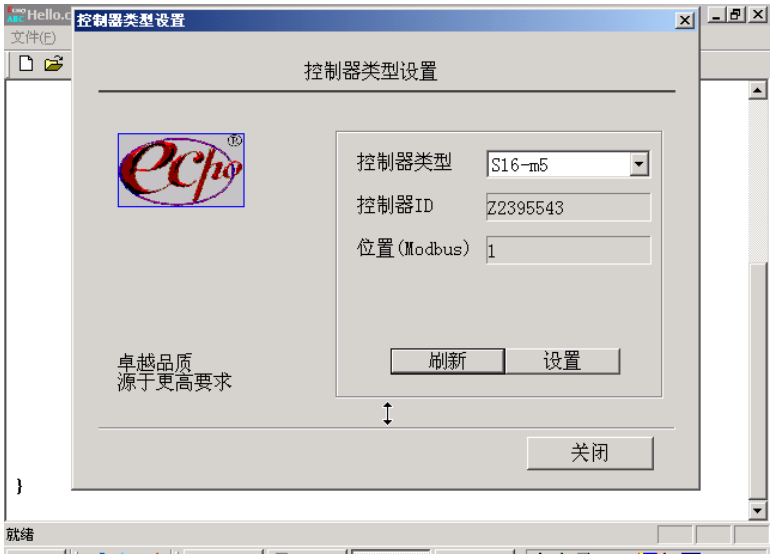



图 6-4

- 点击控制器菜单的初始化控制器命令或工具栏的图标。

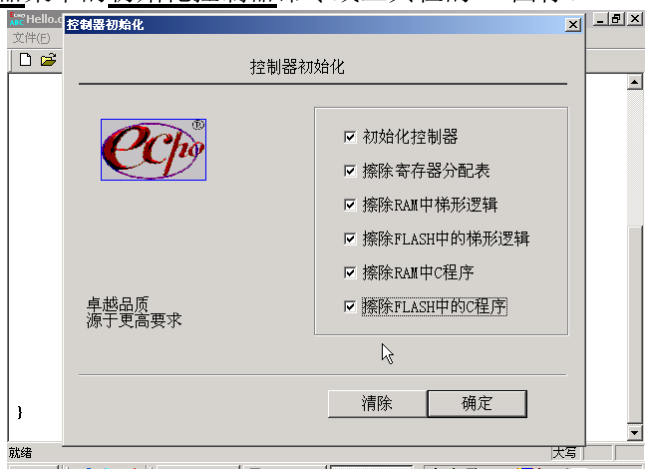



图 6-5

- 选择所有选项：清除梯形逻辑程序、清除 C 程序、初始化控制器与清除寄存器分配列表。
- 点击清除按钮。

图 6-6

此时，控制器可进行编程。

## ➢ 下载程序

下载 hello,world 程序到控制器中：

- 运行 EMC 程序

- 从通讯菜单中，选择 C 程序下载命令或工具栏中选择 图标。

图 6-7

- 进入编辑框中写 HELLO.ABS 程序名。
- 选择下载位置：RAM 或 Flash，注意要与编译时选定的位置一致。
- 点击下载按钮，程序将被下载。

图 6-8

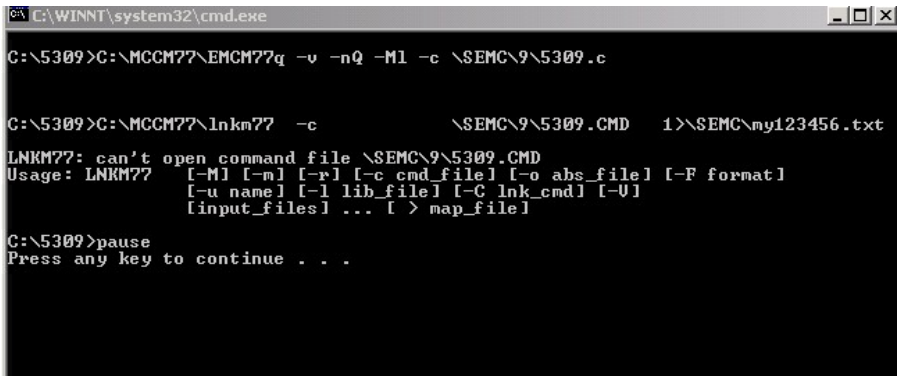
图 6-9

- 从 C 程序下载对话框中点击运行按钮执行程序。Hello,world 信息将输入到 COM 口。

# 第八章 程序开发注意事项

## 8.1 源程序存放目录

编译文件时，出现错误，不能形成.ABS 文件。




### 8.1.1 问题原因

编译器问题，必须将源文件存放到 C:\SEMC 下。就可以正常编译了。

### 8.1.2 纠正此问题

这个问题是 EMC 编译器的问题。它不能被纠正。使用者必须象上面那样绕过此问题。

## 8.2 局部静态变量的使用

编译器为初始化的静态局部变量分配存储区不正确。此存储区是为字符串常量数据专用的。

存储区应当采用初始化变量存储区。

### 8.2.1 问题原因

在 ROM 中的程序不能改变以此方式声明的变量。在 RAM 中的程序可以改变该变量。

变量位于程序内存段，操作系统无法改变变量。修改此变量会导致操作系统认为程序已经被改变。程序能够正确继续执行，但是如果它被 ELadder 的 **C 装载**程序中断或控制器复位后，就不能重新运行。操作系统会检测到程序内存被破坏，因而不能执行程序。

### 8.2.2 例子

编译器在下面例子会产生错误代码。变量的存储被分配到了 string 段，它位于 initvars 段。

如果程序状入了 ROM，那它不能改变变量。

如果程序装到了 ROM，那它在写入控制器内存后可运行一次。所有以后的执行都会失败。

```
Void main()
{
    static int a=1;
    a++;
    /*其它代码*/
}
```

## 8.2.3 问题的解决

有两种方法可以绕过此问题

1、用全局变量代替局部变量。例如：

```
static int a=1;
void main(void)
{
        a++;
        /*其它代码*/
}
```

2、如果局部变量被初始化为零，则无初始化的静态局部变量可以使用。例如

```
void main(void)
{
        static int a;
        a++;
        /*其它代码*/
}
```

此例子中，声明：

static ing a;

等价与：

static int a=0;

运行程序之前操作系统设置没有初始化的变量（存储在 zerovars 段）为零。

## 8.2.4 纠正此问题

这个问题是 EMC 编译器的问题。它不能被纠正。使用者必须象上面那样绕过此问题。